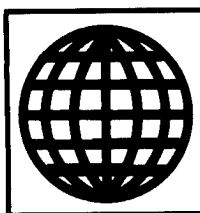
JPRS-CST-92-020
28 October 1992



FOREIGN
BROADCAST
INFORMATION
SERVICE

JPRS Report

Science & Technology

China

High-Performance Computer Systems

Science & Technology

China

High-Performance Computer Systems

JPRS-CST-92-020 *28 October 1992*

CONTENTS

Performance, Evaluation of Parallel Graph Rewriting Abstract Machine PAM/TGR

92FE0850 Shenyang XIAOXING WEIXING JISUANJI XITONG [MINI-MICRO SYSTEMS] in Chinese Vol 13, No 8, Aug 92 pp 1-9

[Article by Tian Xinmin [3944 2450 3046], Wang Dingxing [3769 7844 5281], and Shen Meiming [3088 5019 2494] of the Qinghua University Department of Computer Science and Technology, Beijing: "Performance and Evaluation of a Parallel Abstract Machine for Term Graph Rewriting (PAM/TGR)[*]"; MS received 16 May 92]

[Text] This article begins with a brief description of the design and implementation of a parallel abstract machine for term graph rewriting (PAM/TGR) based on the heterogeneous parallel graph rewriting execution model (HPGREM). It then discusses performance evaluation standards oriented toward the PAM/TGR and describes the possible occurrence of two types of acceleration phenomena, over acceleration and ill acceleration, in parallel multiprocessor systems and provides a definition of over-speedup ratio and ill-speedup ratio. On this foundation, we tested and evaluated the performance of the PAM/TGR based on typical benchmarks. The results of the tests show that the PAM/TGR machine has good acceleration results and a high processor utilization rate, that it can effectively avoid serious ill speedup phenomena, and that its system performance is superior to similar systems in foreign countries at the present time.

Key terms: Parallel graph rewriting model, parallel abstract machine, benchmark program, performance evaluation

I. Introduction

Extremely significant advances have been made in work in many areas in vanguard realms of research on parallel computers, especially the generally acknowledged successful integration of parallel computing technology with artificial intelligence technology, which has effectively spurred the development of parallel processing technology[3]. The development of VLSI technology and various new computing technologies have also provided effective support for constructing high-performance parallel processing systems. Theoretical research on parallel graph rewriting computation technology as a new computing technology can be traced back to the 1930's. At that time, A. Church and other logicians did creative research on theoretical aspects of rewriting computation. During the past several years, extremely significant research achievements in rewriting computation technology in areas like time state logic, program design logic, formalized software development, and so on have aroused interest and attention among people in the computer science field. Rewriting computation theory as the study of basic theory on computability is focused on defining precise symbolic system descriptions and usable mechanical methods for carrying out evaluation operations and data, which fundamentally determines that computing based on rewriting has superior mathematical properties. We used intensive research on graph rewriting computation theory as a basis for proposing an expanded graph rewriting model (EGRM)[12,13] for use in supporting the effective implementation of functional language and parallel logic language. Because the special-purpose hardware designed to support parallel graph rewriting computation is restricted by data relationships among tasks, the hardware is too expensive. Thus, using several commercially available single processors to put together a parallel multiprocessor system with a restructurable topological architecture to support the highly effective implementation of parallel graph rewriting computation has become a very significant research topic. The key to the problem lies in how to compensate for the semantic differences between the rewriting model and the Von Neumann model used in the processors now currently available commercially, so we designed and implemented a new parallel abstract machine for term graph rewriting (PAM/TGR) for use as a interface between the rewriting model and the Von Neumann model[7,9,10]. The PAM/TGR is a parallel computing system that supports multiple types of declarative languages oriented toward artificial intelligence applications and research on it will provide substantial research experience for parallel implementation technologies for declarative languages.

This article first provides a brief description of the parallel abstract machine for term graph rewriting (PAM/TGR) and its architecture and then provides and discusses performance evaluation standards oriented toward PAM/TGR. Part IV of this article describes and analyzes the results of performance tests of the parallel abstract machine PAM/TGR based on benchmarks and concludes by comparing the operational performance of several benchmark programs in the PAM/TGR to the performance of similar systems in foreign countries and offers conclusions.

II. The Parallel Abstract Machine PAM/TGR and Its Architecture

There are two typical abstract machines oriented toward effective implementation of declarative program design languages: 1) The sequential graph rewriting function abstract machine G-machine[1] proposed by L. Augustsson and 2) The sequential logic abstract machine WAM-machine proposed by D. H. D. Warren[2]. Unlike these generally known sequential abstract machines, parallel abstract machines based on different parallel execution models are still in the research stage. It should be noted that the G-machine and WAM-machine cannot simply be expanded into parallel abstract machines that support the corresponding functional programs and logic programs. For this reason, we proposed and

designed the parallel abstract machine PAM/TGR and corresponding architecture based on the heterogeneous parallel graph rewriting execution model HPGREM. Our design goals were:

1. The PAM/TGR should be capable of effectively supporting functional language and logical language computing semantics under a unified framework of expanded graph rewriting computation.

2. Effectively support the heterogeneous parallel graph rewriting execution model.

3. Absorb the advantages of the G-machine, WAM, TIM, <v, G>-machine, FAM, and other abstract machines. Develop parallelism with an appropriate granularity on the basis of fully utilizing single-processor resources.

4. Easy conversion of attributed AND/OR graphs into PAM/TGR instructions.

5. Suitable for effective implementation in distributed memory or shared memory multiprocessor systems.

The parallel abstract machine PAM/TGR is composed of four parts: the memory organization, data representation, instruction set and machine instruction execution algorithm, and multiprocessor architecture[9,11].

A. Memory organization

The parallel abstract machine memory organization is composed of five parts: the Code Space, Data Space, Stack Space, Heap Space, and Register Group.

1. Code space: the executable codes and control information corresponding to these codes generated by the storage compiler.

2. Data space: local solution space used to record the final values and rewriting computation graphs for argument nodes.

3. Stack space: the top nodes and rewritable nodes used to store rewritable sub-graphs and the computing environment of the rewritable nodes.

4. Heap space: used to store the construct nodes, lazy rewriting nodes, and structured data generated by executed graphs.

5. Register group: used to store the stack top pointers for each of the stacks, currently rewritable sub-graph top node pointers, and recoverable garbage space chain tail pointers.

We use the unified name of graph rewriting space for these five types of memory space for the actual memory system of the heterogeneous parallel graph rewriting execution model. The execution graph is the memory object of this system and the graph nodes are the most fundamental units. Actually, each graph node is a complex data structure that is composed of a graph node identifier, operation sub-name, certain parameter fields, and a control field in the form:

Id: Op (Arg_1, Arg_2, ..., Arg_n, Ctl_1, Ctl_2, ..., Ctl_m)

Id is the address of the graph node in the graph space, Op is the node operator, Arg_i is the parameter node, and Ctl_i is the control node. It should be explained that the type of control field used is closely related to the actual execution control model. For the shared graph nodes, however, the control field is essential. In the organization of the parallel abstract machine memory system, we use a low-frequency data copying tactic to compress the rapid consumption of memory space by the independent environment during the rewriting task execution process. A pointer re- initiation and copying tactic is used for complex data structure citation and small granularity non-moving tasks to reduce environment copying overhead and to always place the overall computing environment for rewritable tasks directly in the heap space pointed out by the current environment stack stack-top pointer according to the category of parameter node, thereby improving rewriting efficiency.

B. Data representation and graph node format

The data in the parallel abstract machine is composed of two fields: an indicator field and value field. The indicator field stores the data category and the value field stores a value or pointer depending on the category identification in the indicator field. The data categories in the PAM/TGR are: integer, real type, Boolean, character, character string, atomic, null atomic, construct, user-defined construct, tuple, and argument. See reference [11] for a detailed definition of the storage format for each data category.

It should be explained that the graph node formats CONS, CONST, REF, and OPR are category indicator classifications in the data representation. They have a more regular meaning and are a type of classification tag. This type of classification aids in graph space allocation and scratch area recovery.

C. Parallel abstract machine PAM/TGR instruction set

The instruction set of the parallel abstract machine contains 10 kernel instructions. Declarative programs use multilevel programs for conversion and translation into the PAM/TGR instruction sequence. Finally, we converted this instruction sequence into execution object codes that can be executed in a multiprocessor system composed of 16 Transputers. See references [9] and [11] for the PAM/TGR instruction form and semantic definitions. Theoretically, the parallel abstract machine PAM/TGR is suitable for implementation in shared memory multiprocessor systems and is also suitable for implementation in distributed memory multiprocessor systems. The only difference is the difference in synchronous implementation modes for parallel tasks. For the data expressions corresponding to the parallel abstract machine PAM/TGR, the abstract machine instructions have six types of high-level addressing modes, as described in reference [11]. Adoption of these six types of high-level addressing modes enables the compiler to process scalar data, structured data, and rewriting closures using a simple unified model. It should be explained that the instruction set for the parallel abstract machine PAM/TGR does not include

instructions related to task requests and inspection of this type of request because the execution behavior of these two instructions does not affect the state of abstract machine execution. The state of each stack and the state of the heap space are not affected, whereas during operation the task dynamic management system performs the function of inspecting resource states and distributing parallelable tasks. The abstract machine instruction execution algorithm describes the state conversion of the abstract machine and simultaneously directly maps the abstract machine instructions in the mapping algorithm of its architecture[7,11].

D. The architecture of the parallel graph rewriting abstract machine PAM/TGR

Parallel multiprocessor architectures can generally be divided into two categories. One category is loosely-coupled distributed memory multiprocessor systems in which each processor communicates via an interconnection network at a communication speed that is determined by the network bandwidth. The second category is tightly-coupled shared memory multiprocessor systems in which the communication speed among processors is determined by the memory bandwidth. Both of these types of multiprocessor systems pose the problem of network delays or memory delays due to the communication or memory bandwidth in development of computing parallelism to support parallel graph rewriting execution models. If fine granularity parallelism is being developed, it is quite possible that the large amount of accessing, communication, and environment copying that result may offset the benefits from developing computing parallelism. As a result, the design of a multiprocessor architecture to develop coarse granularity activated conservative parallelism based on a heterogeneous parallel graph rewriting execution model adopts a distributed memory multiprocessor architecture[11]. The parallel abstract machine PAM/TGR is a loosely-coupled multiprocessor system composed of several processing elements [PE] with local memory via an interconnection network. By cooperative operation of the task distributor (TD) on each PE, the execution graph is distributed in a passive manner to each idle PE and each PE carries out rewriting and conversion of each of the rewritable subgraphs that are distributed to the local graph memory space. The local graph memory space in each PE in the PAM/TGR independently compiles and rewrites the required parameter environments that are obtained via the interconnection network in a message passing mode. In an Eager computing mode, the parameter environment is always distributed to the corresponding PE along with the rewrite nodes (rewriteable nodes). The organizational model of the PAM/TGR is a fully distributed model, which is entirely different from the previously constructed virtual global memory space supported graph rewriting computation model. This model can avoid the problem of environmental consistency implicit in shared memory space and effectively support the dynamic distribution of execution graphs.

This section provides a brief description of the design of the parallel graph rewriting abstract machine PAM/TGR and its architecture. The high-level instruction set of the PAM/TGR is clear and simple, has strong applicability, and is capable of effectively supporting heterogeneous parallel graph rewriting execution models. We have implemented the PAM/TGR machine in a hardware environment constructed from a Transputer array. Moreover, the PAM/TGR is a scalable multiprocessor system with a topological architecture that can be restructured by using programs.

III. Performance Evaluation Standards for the Parallel Graph Rewriting Abstract Machine PAM/TGR

Performance testing and evaluation for the parallel multiprocessor system is an important aspect of its system performance. There is an endless stream of parallel computer systems now being promoted internationally, for example the MARK-II based on the Intel 80286/80287, the MARK-III based on the Motorola 68020/68881, and the distributed memory system NCUBE 6400 series promoted by the NCUBE Company in the past few years (maximum of 8,192 processing elements). As these multiprocessor systems have appeared, a series of questions like how to test and evaluate these systems, how to exploit system potential and determine their scope of applications, and so on have become increasingly important. In general, testing and evaluation methods for parallel multiprocessor systems can be divided into three categories: 1) Hard testing methods for testing hardware performance; 2) Theoretical analysis methods based on abstract models; 3) Soft testing methods based on benchmark programs. Hard testing usually only provides component performance and is a theoretical peak value. Thus, its actual applications value is not great. The theoretical analysis method based on abstract models mainly uses simulation analysis and probability statistics to obtain communication performance, response time, and other theoretical values. These theoretical values play a substantial guiding role in actual work but are often quite different from the actual performance. Thus, people usually use a soft testing method based on benchmark programs. The soft testing method is simple and direct and has been universally adopted for this reason. We used this type of soft testing method in performance testing and evaluation of the parallel graph rewriting abstract machine PAM/TGR.

There are quite a few indices for evaluating the performance of parallel systems and the scope of evaluations is very broad. Their main differences from sequential systems are the use of time overlapping, resource sharing, and other modes to increase program execution efficiency. The parallel graph rewriting execution system places no added burden on users when using this system and the inherent parallelism in the program is automatically developed by the PAM/TGR and the inherent sequentialness is automatically protected by the PAM/TGR. As a result, ML and PARLOG programs on sequential machines can operate on the PAM/TGR machine without revisions, and there are substantial acceleration benefits. We tested and evaluated the PAM/TGR system in execution time, speedup ratio, execution efficiency, rewriting speed, and other areas. The related basic concepts are defined below:

Definition 3.1: Execution time

Execution time is the time spent in executing program P in a certain parallel environment, and is recorded as T (in seconds).

Definition 3.2: Speedup

Assuming that for program P, T_i is the execution time of program P with i processors, the speedup ratio S_n for n processors is defined as $S_n = T_1/T_n$.

Definition 3.3: Execution efficiency

The execution efficiency of program P in n processors is $E_n = S_n/n$, and the size of E_n is a reflection of the utilization rate of the n processors.

Definition 3.4: Rewriting speed

Assuming that for program P, R is the rewriting steps in the process of executing program P, the rewriting speed for n processors is $R_n = R/T_n$ (rewriting steps/second).

In these four indices, the speedup ratio can be divided into three situations: over acceleration, normal acceleration, and ill acceleration. These are defined as:

Definition 3.5: Over speedup

Assuming that for program P, if the speedup ratio for program P in n processors is $S_n > n$, then $E_n > 1$ gives an over speedup ratio $O_n = (S_n/n)-1$

Definition 3.6: Normal speedup

Assuming that for program P, if the speedup ratio S_n for program P in n processors satisfies $1 < S_n < n$, then $0 < E_n < 1$, so the normal speedup ratio is $N_n = S_n$.

Definition 3.7: Ill speedup

Assuming that for program P, if the speedup ratio in n processors is $S_n < 1$, then the ill speedup ratio is $I_n = 1 - S_n$.

Over speedup can only appear accidentally under two conditions: 1) When solving search problems and NP problems for artificial intelligence, multiple processors computing in parallel accidentally find the shortest solution path. 2) When program P is operating in a single processor, because of memory capacity limits, over acceleration can occur in a situation in which the overhead is very large due to the processor frequently carrying out space recovery and reallocation when multiple processors are operating and each processor expends very little overhead in this area. This condition occurred in testing the PAM/TGR. The ideal speedup ratio of parallel programs in n processors is n and the ideal speedup ratio of sequential programs is 1. However, because of restriction by the communication, synchronization, and other added overhead in developing computing parallelism, and restriction by the inherent parallelism of the algorithm and the program itself, the speedup ratio of parallel programs is generally in the range (1,n). The speedup ratio for sequential programs is $0 < S_n < 1$.

Figure 1 is a diagram of the speedup ratio regional distribution for 16 processors.

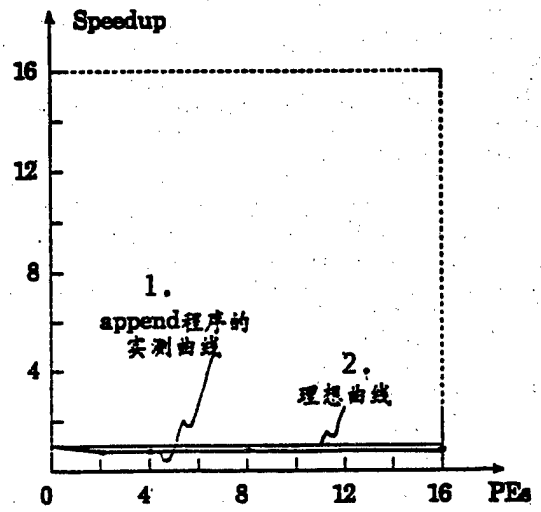


Figure 1. Speedup Ratio Regional Distribution

Key: 1. Measured curve for append program; 2. Ideal curve

In the figure, A_0 is the normal acceleration region, A_1 is the over acceleration region, and A_2 is the ill acceleration region. A relatively ideal parallel system would have these characteristics when executing the chosen benchmark: 1) A benchmark with good parallelism can produce a good speedup ratio; 2) Sequential codes without parallelism can be executed with high efficiency in parallel systems according to their execution models in sequential systems; 3) Development of parallelism can produce a relatively good balance between the traffic and amount of computation and develop parallelism with a rational granularity. We described above several standards for testing and evaluating the performance of parallel multiprocessor systems. The selection of a rational and effective benchmark based on these standards is very important for making a fair evaluation of the performance of a parallel computer system.

IV. PAM/TGR Performance Testing and Evaluation Based on Benchmarks

Evaluation of the performance of parallel computer systems and sequential computer systems requires that the benchmarks selected be substantially representative and that they at least be capable of representing a group of identical or similar computing problems and not involve certain special cases, so that the conclusions drawn from these benchmarks can be extended naturally in applications to similar problems. Based on this type of guiding ideology and focusing on execution time, speedup ratio, execution efficiency, and other aspects, and taking into consideration typical programs for parallel systems for oriented declarative programs used for testing in foreign countries, we selected the fib, prime, hanoi, queen, tak, qsort, and other typical programs as the programs for testing the PAM/TGR system. Table 1 lists the results of performance tests for the corresponding benchmarks in the PAM/TGR system and Figures 2(a) to 2(d) plot the acceleration curves for several test programs run on the PAM/TGR.

Table 1. Results of Testing Several Benchmark Programs in the PAM/TGR

Benchmark program	T_1 (seconds)	T_4 (seconds)	T_8 (seconds)	T_{16} (seconds)	S_{16}	E_{16}	R_{16}(rewriting steps/second)
fib(30)(M)	39.6084	15.2075	6.716	4.2409	9.34	58.375%	1373K
fib(37)(M)	1148.0182	438.5607	168.0175	104.1888	11.02	68.875%	1623K
prime(100000)(M)	26.6839	7.7211	3.9589	2.0267	13.17	82.31%	987K
prime(200000)(M)	61.3976	18.0924	9.2527	4.6899	13.09	81.81%	853K
hanoi(20)(P)	121.9000	30.5136	15.2879	8.0828	15.08	94.25%	778K
hanoi(21)(P)	243.7969	60.9885	30.5328	15.8271	15.40	96.25%	795K
queen(8)(P)	20.8886	12.0197	2.5026	2.4459	8.5402	53.37%	307K
queen(9)(P)	183.9586	100.0671	20.3672	10.0980	18.22	113.85%	368K

Note: M in the table represents the use of the ML program in the test while P represents the use of the PARLOG program in the test.

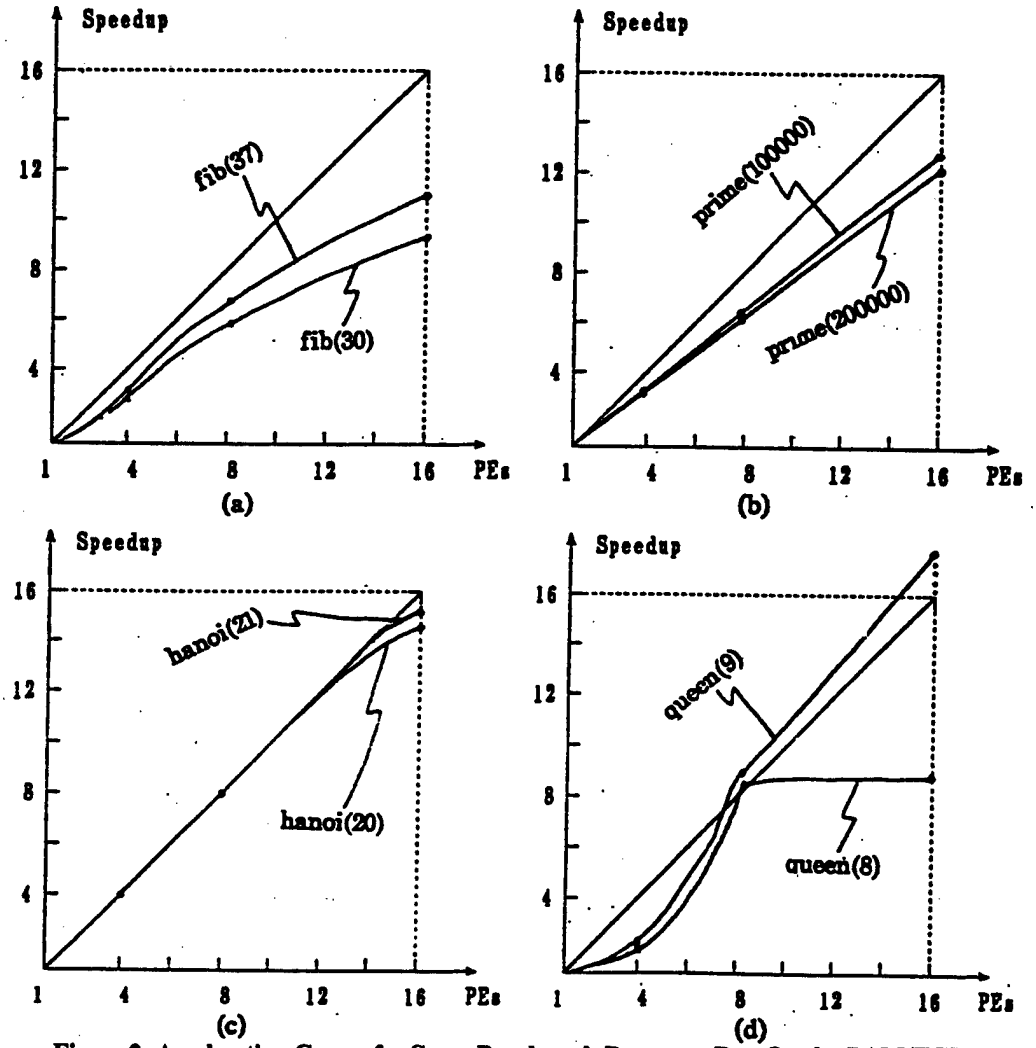


Figure 2. Acceleration Curves for Some Benchmark Programs Run On the PAM/TGR

In testing the PAM/TGR system, the benchmarks we selected like the queen problem, prime problem, (fanta) problem, and so on are all actual problems in the fields of mathematics and AI, and all of them have specific computing scales and time and space consumption. This is especially true of the implementation of descriptions based on functional and logical program design languages, which place rather high requirements on effective management of memory and development and control of parallelism. From another side, they can also be used to examine the PAM/TGR system's automatic development and management of computing parallelism. The test results to examine the PAM/TGR system performance show that the rewriting speed for the several benchmark programs attained relatively high indices.

It should be noted that over acceleration phenomena occurred during the solution process for the 9 queen problem described by the PARLOG program. The reason was that the internal memory of a single Transputer could not satisfy the requirements for solution of the 9 queen problem, which increased the dynamic recovery and reallocation operations for much of the memory space. In the multiprocessor system composed of 16 Transputers, because the computing scale for each of the subtasks was relatively small, the memory space utilization problem was alleviated, which avoided additional overhead due to large amounts of garbage space recovery and reallocation, which in turn led to the occurrence of over acceleration phenomena. The over speedup ratio $O_n = (17.662/16)-1 = 0.102$. In addition, because the solution process for the queen problem involved a large amount of frequent transmission of dynamic data structures, the actual speedup of the program when operating in the PAM/TGR was substantially poorer than the ideal speedup ratio. The results of research by Augustsson and Johnson, et al. show that the actual speedup ratio in a parallel system with a processor scale of 15 to 20 PEs is generally between 5 and 11. The results of our research indicate that the actual speedup ratio of the typical programs in the PAM/TGR (16 PEs) was generally between 6.38 and 15.57. The 9 queen program for the parallel logic program PARLOG was an exception in that it involved over acceleration with a speedup ratio of 18.22. In addition, we have listed below the results of tests of an actual example of a logical language PARLOG table append program to test whether or not there is a serious ill speedup ratio in the parallel system, as shown in Table 2 and Figure 3.

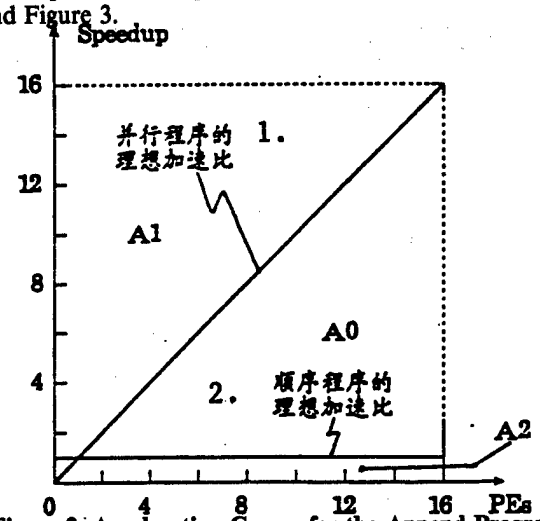


Figure 3. Acceleration Curves for the Append Program Run On the PAM/TGR

Key: 1. Ideal speedup ratio for parallel programs; 2. Ideal speedup ratio for sequential programs

Table 2(a). Execution Time for the Append Program Run On the PAM/TGR

Benchmark program	Execution time				
	T_1 (seconds)	T_2 (seconds)	T_4 (seconds)	T_8 (seconds)	T_{16} (seconds)
append(50,100)	0.0454	0.0529	0.0537	0.0592	0.0532
append(100,200)	0.2828	0.3272	0.3892	0.3938	0.4041

From the definition $I_n = 1 - S_n$ we can derive the following table:

Table 2(b). Ill Speedup Ratio for the Append Program Run On the PAM/TGR

Benchmark program	I_n			
	I_2	I_4	I_8	I_{16}
append(50,100)	0.142	0.115	0.233	0.147
append(100,200)	0.136	0.273	0.282	0.300

Figure 3 and Table 2(b) show that the append program in the PAM/TGR basically maintains its sequential computing efficiency, which is an indication that the PAM/TGR has a very small negative role in sequential program execution. It also illustrates that the optimized compiler in the PAM/TGR is capable of effectively recognizing parallelism and generating optimized abstract machine instructions based on the inherent characteristics (computing granularity and data interrelatedness restrictions) of the program.

V. Comparison With the Performance of Similar Systems in Foreign Countries and Conclusions

Parallel multiprocessor systems are now being widely used in all applications realms and their scalable processor scale, high memory bandwidth, and high communication bandwidth all substantially improve the performance of parallel multiprocessor systems. Nevertheless the network delays caused by cooperation among processors and the resource competition problem still exist. We gave full consideration to these problems in developing the PAM/TGR and adopted a variety of optimization technologies including development of coarse granularity for conservative parallelism, partial scheduling analysis during compiling, and so on[7,8,14] to enable a substantial improvement in the performance of the PAM/TGR. To further test the performance of the PAM/TGR system, we made comparisons of the PAM/TGR system with similar systems internationally based on typical programs. The results of the comparisons are shown in Table 5.

Table 5. Comparison of the Performance of Several Benchmark Programs Run On the PAM/TGR With the Performance of Similar Systems in Foreign Countries

Benchmark program	System name				
	Alfalfa	Buckwheat	APEX	K-LEAF	PAM/TGR
	Hardware environment				
	17Intel 80286 (hypercube)	12NS32032 Encore Multimax	20 Sequent Balance 21000	16T800	16T800
pfac(1,100)	1.90s	0.29s(F)	-	-	0.058s(F)
nqueen(8)	7.50s(F)	2.05s(F)	-	-	1.473s(F)
qsort(200)	9.00s(F)	2.35s(F)	-	-	0.320s(F)
tak(18,12,6)	-	-	7.20s(L)	-	2.710s(L)
hanoi(15)	-	-	6.47s(L)	-	2.287s(L)
fib(29)	-	-	-	7.750s(F)	3.29s(F)
nqueen(8)	-	-	-	3.080s(L)	2.721s(L)
nqueen(9)	-	-	-	13.937s(L)	10.378s(L)

Note: F: represents a functional program, L represents a logic program, time units are in seconds.

Table 5 [tables 3 and 4 omitted in original text] shows that the PAM/TGR system performance is very good. It can support highly efficient execution of functional language (ML) programs and parallel logic language (PARLOG) programs and its execution speed is significantly superior to similar systems in foreign countries. The Parallel Military Information Processing System and cryptographic system developed on the PAM/TGR by the Third Department of the [PLA] General Staff Computing Central Station show that the PAM/TGR system has definite applicability. It has a friendly user interface and convenient program debugging. Our further research work will adopt an even more advanced processor chip (such as the Transputer T9000 chip) and more advanced network communication technology, such as the Wormhole pathfinder technology. The adoption of SVM technology on this foundation to construct a new type of shared-distributed memory (SDM) system will effectively reduce network delays and utilization of processor resources, which will further improve the performance of the parallel abstract graph rewriting machine PAM/TGR.

References

[1] L. Augustsson, Compiling Lazy Functional Language Part II, Ph.D. Thesis, Department of Computer Science, Chalmers University, Sweden.

[2] P. G. Bosco, C. Cecchi, and C. Moiso, An Extension of WAM for K-LEAF: A WAM Based Compilation of Conditional Narrowing, in Proceedings of the 6th Conference on Logic Programming, October 1989 pp 318-333.

[3] B. S. David, Architecture-Independent Parallel Computation, IEEE Computer, Vol 23, No 12, December 1990 pp 38-50.

[4] J. Fairbairn, TIM-A Simple Lazy Abstract Machine To Execute Supercombinator, in Proceedings of the IFIP Conference on Functional Program Languages and Compiling Architecture, Springer Verlag LNCS 274, pp 34-45.

[5] T. Johnsson, Parallel Graph Reduction With the <v, G>-Machine, in Proceedings of Functional Program Languages and Compiling Architecture, 1989 pp 202-213.

[6] X. M. Tian, D. X. Wang, M. M. Shen, and W. M. Zheng, An Efficient Compiling Implementation of CIL Rewriting Language on Multiprocessor System, Technical Report, Qinghua University, October 1991.

[7] X. M. Tian, D. X. Wang, M. M. Shen, and W. M. Zheng, A Practical Eager- Lazy Control Method for

Dynamic Deriving Parallel Tasks, accepted by Journal of Software, 29 November 1991, China.

[8] X. M. Tian, D. X. Wang, M. M. Shen, and W. M. Zheng, The Compile-Time Partial Scheduling Strategies for Optimizing Granularity of Parallel Graph Rewriting, accepted by Journal of Computer Science and Technology, February 1992.

[9] X. M. Tian, D. X. Wang, M. M. Shen, and W. M. Zheng, The Parallel Graph Rewriting Abstract Machine and Its Efficient Execution Mechanism, in Proceedings of the 1992 National Intelligent Information Technical Conference, April 1992.

[10] X. M. Tian, D. X. Wang, M. M. Shen, W. M. Zheng, and Dongchan Wen, An Optimized Parallel Compiler for Executing Declarative Languages of Transputer Array, in Proceedings of 15th Occam/Transputer International Conference, 12 to 15 April, 1992.

[11] X. M. Tian et al., Parallel Graph Rewriting Abstract Machine and Its CIL Compile-System, Technical Report 863-306-101 (3), Qinghua University, May 1992.

[12] D. X. Wang, Specification of CIL, Technical Report, Department of Computer Science and Technology, Qinghua University, Beijing, China, January 1989.

[13] D. X. Wang, a High-Level Compiling Implementation of PARLOG Based on Extended Graph Rewriting, in Proceedings of Tools for Artificial Intelligence 90, Washington, D. C., November 1990.

[14] D. X. Wang, Parallel Graph Reduction Intelligent Workstation, Research Report 863-306-101 (1), Qinghua University, May 1992.

[*]Additional authors of this article were Zheng Weimin [6774 0251 3046], Wen Dongchan [3306 0392 1292], and Xiong Jianxin [3574 1696 2450]. This research project was funded by the State 863 High Technology Project 863- 306-101 and the State Higher Education Ph.D. Disciplinary Focus Special Topic Scientific Research Fund 0249136.

Software Development for 980-STAR Systolic Array Computer System

92FE0867A Beijing JISUANJI YANJIU YU FAZHAN [COMPUTER RESEARCH AND DEVELOPMENT] in Chinese Vol 29, No 2, Feb 92 pp 58-62

[Article by Liu Risheng[*] [0491 2480 0581] of the China State Shipbuilding Corporation's Wuhan Institute 709, Wuhan: "Software Development for the 980-STAR Systolic Computer System" [see early brief report in JPRS-CST-90-012, 18 Apr 90 p 16]; MS received Jan 91]

[Excerpts] **Abstract:** This article describes the software functions, architecture, and user interface of China's first Systolic Computer System and concludes by pointing out the direction of further research in the future on this system software. [passage omitted]

I. Overview

The rapid development of modern science and technology continually require the appearance of processors with faster speeds. The systolic technology proposed by professor H. T. Kong at Carnegie-Mellon University in the United States in 1978 appeared in this kind of situation.

Systolic arrays are usually expressed as arrays composed of a number of processing elements (PEs) with identical logic functions; the PEs are connected locally in a simple regular communication geometry. This type of architecture is particularly well-suited to the implementation of VLSI [very large-scale integration].

Systolic arrays are suitable for algorithms with relatively good regularity like image processing, signal processing, matrix operations, etc. Because these algorithms often take up large amounts of time for computing tasks, the use of systolic arrays can greatly increase the overall speed of computing tasks. This characteristic of systolic technology makes it especially suitable for military applications, and it has attracted the attention of the military in the United States as a result. In 1983, the United States Department of Defense listed systolic technology as one of the basic technological projects in its "Strategic Computing Program."

The fixed connections among the component elements of systolic arrays limit the scope of their applications, and as a result people have used program control to change the communication geometry among the elements. This type of array is called a programmable array. A typical example is the Warp Computer developed by Carnegie-Mellon University in 1986. It is a linear array composed of 10 Cells and its 32-bit speed can reach 100 MFLOPS [million floating-point operations per second]. Another representative is the Matrix-1 Computer developed by the Saxpy Company in the United States in 1987. It is a linear array composed of 32 elements and its 32-bit speed can reach 1,000 MFLOPS.

China's first systolic computer is the 980-STAR Computer successfully developed by the China State Shipbuilding Corporation's Wuhan Institute 709 in July 1989. It is a two-dimensional programmable array composed of 4 X 4 elements and its fixed-point 8-bit computing speed can reach 160 MIPS.

Because systolic arrays are only suitable for several relatively regular computations, they are not suited to doing other computations or to system management and software development, and other work. Thus, these arrays often serve as a auxiliary machines connected to a common computer. Users use the systolic array processors via the host machine, and the host machine is responsible for developing applied software, system management, and normal operations for the array processors. Moreover, an appropriate part of the array

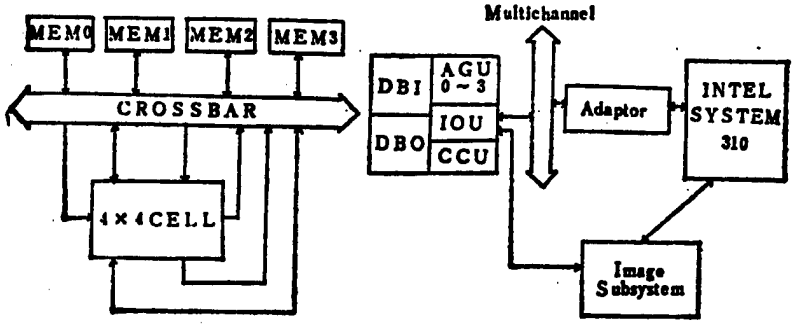


Figure 1. 980-STAR Hardware Structure

processor operations are turned over to the systolic array processors. For example, Warp computers use a number of SUN-3 workstations connected via a network (TCP/IP) as host computers. The Matrix-1 uses a special high-speed channel for connection to a VAX machine. The host machine for the 980-STAR uses an Intel Corporation System 310 microcomputer and the operating system is iRMX86.

Below are described the primary characteristics of the system software for a computer system composed of a host machine and array processors.

II. Architecture

The hardware structure of the overall system is illustrated in Figure 1. The host machine is connected to the array processor system via a Multichannel. In the array processor portion, besides a 4 X 4 systolic array, there is also an interface machine and four memories MEMO-MEM3. The CROSSBAR is the hub for data exchange among the three. The interface machine includes AGU, IOU, and CCU. AGU0-AGU3 are, respectively, the addresses generated for MEM0-MEM3. The IOU passes through the data buffer input DBI and data buffer output DBO to exchange data with the host computer, and it can exchange image data with the graphic subsystem. The CCU is the controlling unit for the array processors. It interprets the instructions from the host machine and controls the coordinated operation of the other processing units. The 4 X 4 Cell executes the actual systolic instructions. Thus, the array processor area has a total of 22 processors. In the host machine area, besides the host computer there is also an image subsystem. After digitization of the image that has been photographed by the camera, it transmits it into the DBI, and after the image is processed by the array processors it can be transmitted into the DBO. The image subsystem then extracts the data from the DBO and transmits it to the display. Of course, the host computer can also extract the resulting data from the DBO.

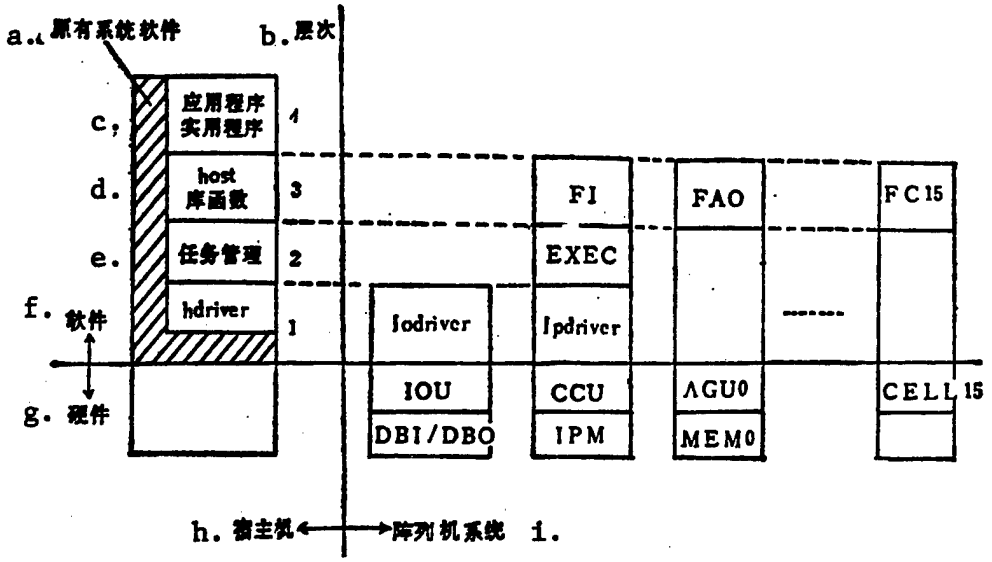


Figure 2. System Software Structure

Key: a. Original system software; b. Level; c. Application programs, utility programs; d. Host library functions; e. Task management; f. Software; g. Hardware; h. Host machine; i. Array machine system

The system software of the overall system is distributed in one host machine and 22 processors, mainly in the host computer. The primary functions of the system software are: 1) Manage the operation and communication of the processors; 2) Provide a user interface to the host machine and support the running of applications software; 3) Provide a user development tools environment in the host machine.

The system software in the array processors is a small amount of microcode implemented in the hardware. In the host machine, to make use of the original operating system and software tools, the original system must be expanded. The structure of the overall system software is illustrated in Figure 2.

The software of the host computer is divided into four levels. The lowest level (hdriver) is the communications program targeted at the Multichannel. The second level is task management, including memory allocation in the array processors. The third level is the host library functions. They are the interface between user applications programs and the array processors. They convert user requests into the data required by the various processing units in the array computer system, and then transmit them in a fixed format to the CCU. The data in this fixed format is called the Function Control Block (FCB). The highest level is the user applications program and the software tools to support array computer software development.

The software of the array computer system is divided into three levels which correspond, respectively, to the three lowest levels of the host machine. Because there is a master-slave relationship between the host computer and array processors, the host computer is the main controlling area, so there is no fourth level for the software in the array processor area. At the lowest level, the IOdriver in the IOU and the IPdriver in the CCU are responsible for transmitting data to the hdriver in the host machine. At the second level, the EXEC in the CCU separate out the FCB from the data that are transmitted in and process the data itself, interpret the meaning of each field in the FCB, and allocate the third level library functions in each processor to complete the work for each one. These library functions are: FI running in the CCU, FA0-FA3 running, respectively, in AGU0-AGU3, and FC0-FC15 running, respectively, in CELL0-CELL15. Thus, each host function call in the host computer area is served by 21 library functions in the array processor area.

The host library functions and array processor functions do not belong entirely to the system software. They are related to actual applications. Thus, the host library functions must be connected to the applications programs prior to running, while the array processor library functions must also be stored as needed in the file and then loaded into each of the processors in the array computer.

When an applications program is running in the host machine, it calls host library functions and the latter convert user requirements into FCB. The FCB move downward level-by-level and at the lowest level pass along hdriver→Iodriver→Ipdriver and are then transmitted to the EXEC which after interpreting them decomposes them and transmits each piece of data into each library function at the three levels for execution.

III. User Interface

Users have two utilization modes for the 980-STAR: the instruction mode and the programming mode. By using the former mode, users can type in instructions and call library functions at the terminal keyboard and instruct the array processors to execute them. In the latter mode, users call the host library functions in their applications programs to request that the array processors complete the specified functions. There are also two processing modes for each of the utilization modes, the line-by-line processing mode and the batch processing mode. When using the line-by-line processing mode, the array processors must return control to the host machine when calling each corresponding host library function to execute each line and wait until the next function is called. In the batch processing mode, however, the user first writes a series of host library function calls into the file. These function call statements can use key words to construct various types of control structures like cyclic statements, conditional statements, and so on. Then users can use a special instruction (Ksubmit) to convert the library functions called in this file into a batch of FCB and they form the corresponding FCB chains according to the syntax in the file. The FCB chains are transferred together to the EXEC in the CCU for interpretation and execution. This file is called a batch processing file. The illustration below shows the four utilization modes for the 980-STAR:

Utilization mode	Instruction mode	Line-by-line processing mode
		Batch processing mode
	Programming mode	Line-by-line processing mode
		Batch processing mode

The instruction mode is easy to use and convenient for program debugging. The programming mode supports program operation. The batch processing mode is easy for users to use and aids in rapid program structuring and debugging. Because the programs in the batch processing files are in the form of the original programs, they do not have to be compiled, so they are very easy to compose and revise. Particularly important is that the batch processing mode is especially suited to real-time off-line applications. In such situations, the array processors can be independent of the host machines after they receive the FCB chains transmitted from the host computer and carry out direct data transmission with

peripheral real-time equipment, which greatly accelerates the response of the array processors to peripheral signals.

[passage omitted]

IV. Further Work

The 980-STAR systolic computer system described above passed examination and acceptance by the National Defense Science, Technology, and Industry Commission in July 1989 and has been operating in an excellent manner in machines to date. This is a principle prototype that is distant from application and requires much more work. We must continue working in the following areas in relation to the system software:

A. Software development tools

This system has already provided several host library functions and the corresponding array processor functions regarding image and signal processing and matrix operations. This is not enough for different types of applications. Although the 980-STAR is a programmable systolic computer, it is still hard for normal users to compile new library functions, so users must be provided with several software development tools for them to conveniently generate the library functions they require. Some work is now being done in this area.

B. Convenient applications interfaces

The user interfaces now provided by the system are mainly targeted at programmers and several special interfaces targeted at certain specially determined fields must be provided.

C. Support for various types of operating systems and high-level languages

At present this system can only run in an iRMX86 operating system and utilize PL/M-86 high-level language.

D. Support for even higher-grade host computers to take full advantage of array processor capacity

E. Improve system software operating efficiency

Thanks: The author would like to offer his gratitude for the warm support and close cooperation received from comrade He Guo [0149 0948] and comrade Kang Hongsheng [1660 1347 3932] in the process of developing software for the 980-STAR system. Program implementation was mainly done by Wu Hua [0702 2901] and other comrades.

[*]Liu Risheng was born in 1944 and graduated from China University of Science and Technology in 1967. He received a Master's degree from the Chinese Academy of Sciences Computing Institute in December 1981 and is a senior engineer. He is mainly involved in research on operating systems, UNIX systems, and real-time control systems. The achievements described in this article received a first-place S&T progress award from the China State Shipbuilding Corporation in 1990.

References

[1] M. Annarotone et al., Warp Architecture: From Prototype to Production, AFIPS Conference Proceedings, Vol 56, 1987 pp 133-140.

[2] B. Bruegge et al., The Warp Programming Environment, ibid., pp 144-148.

[3] D. E. Fourlser et al., The Saxpy Matrix-1: A General-Purpose Systolic Computer, IEEE Computer, July 1987 pp 35-43.

Parallel Processing System Using RISC Technology

92FE0867B Beijing JISUANJI YANJIU YU FAZHAN [COMPUTER RESEARCH AND DEVELOPMENT] in Chinese Vol 29, No 5, May 92 pp 50-56

[Article by Chen Renfu [7115 0083 3940] (deceased) and Xu Youhui [6079 0147 6540][*] of the East China Institute of Computing Technology, Jiading, Shanghai: "Parallel Processing System Using RISC Technology"; MS received Aug 90]

[Excerpts] **Abstract:** This article discusses a parallel processing system constructed from a RISC [reduced instruction set computing]-chip computer and interconnection modes, and on this basis focuses on implementation of system control, management, communication, and so on under a XENIX environment. It concludes with an evaluation of the performance of this system using standard testing programs and a high-order matrix computation program.

I. Introduction

[passage omitted]

The sections below describe the architecture of the parallel processing system (abbreviated as the ECI-PPS [East China Institute-Parallel Processing System]), its software application environment, an evaluation of overall system performance, and some of its characteristics.

II. RISC T800 Architecture

Before introducing the architecture of the ECI-PPS, we must first provide a brief description of the T800 as an important part of this system.

The T800 is a RISC processor that is being promoted by the INMOS Company. In this system we use a 20 MHz chip with fixed point operating speed of 10 MIPS and floating point operating speed of 1.5 MFLOPS. Figure 1 [not reproduced] illustrates the components of the architecture.

The system service units perform reset and total control functions for all units on the chip. The 64-bit floating point unit and 32-bit processor are used to do floating

point operations and various other types of operations. The FPU can execute single and dual-precision floating point arithmetic and it conforms to ANSI-IEEE 754-1985 standards. Its external memory interface can access 4GB of address space and there is 4KB of unified addressing space on the chip. The interconnection server is the part provided by this chip that is used for interconnection among the controller chips. The parallel processing system constructed from the T800 can be used in scientific/engineering computations, graphics/image processing, real-time processing, and other realms.

III. ECI-PPS Architecture

A. ECI-PPS design principles

The design principles for the ECI-PPS employ a building block-type modularized idea that uses additional processors on the host computer to implement parallel processing functions for the system as a whole. The main task of the host machine is responsibility for management of the overall parallel system, whereas key technologies for the ECI-PPS are composed of the processor array in the parallel processing system that is actually responsible for executing user jobs, effectively organizing multi-element processors into an organic processing array, and coordinating the completion of user jobs (including job allocation, scheduling, etc.), and the interface technology between the processor array and host machine.

We mentioned in the previous section that the T800 chip has four interconnection channels and that each interconnection channel has a pair of input/output channels. These channels are used for interconnections among chips, so this type of chip can be used to put together an architecture on an arbitrary scale. The ECI-PPS uses exactly this type of chip and there is substantial flexibility in the combinations of architectures. Moreover, the system composed of INMOS Company chips provides relatively complete tools such as parallel language compilers and some debugging tools.

Each processor in the system should have its own local memory because when it is doing large numbers of numerical computations, stubborn use of a shared memory arrangement will inevitably lead to problems like an increase in processing units, bus overburdening, and so on.

The parallel processing array has one processor that serves as a root processor. Its main responsibility is communication between the processor array and the host computer. All of the data in the processor array must go through this root processor to communicate with the host machine. The reasons are one, that it reduces the burden on the bus, and two, that it reduces control.

The basic interconnection arrangement for the processor array is a pipelining interconnection mode that utilizes one pair of input/output channels. If all three of the other channels have a completely hard-wired interconnection, the topological architecture is fixed and is not flexible enough. Thus, the adoption of a basic pipelining architecture with an added interconnection network enables the production of this type of topological architecture.

Based on present applications requirements and development trends, when providing a software environment the operating system of the UNIX series should be the primary environment. Here, we chose the UNIX variant operating system XENIX System V, but in taking into consideration the large number of DOS users at the present time we also provided a DOS environment for the parallel processing system.

B. ECI-PPS architecture

The ECI-PPS is composed of a host computer, a processor array, and an interconnection network. The architecture is illustrated in Figure 2.

In the diagram, the host computer is the controller machine for the overall parallel processing system and it manages user input/output operations like printing, screen output, keyboard input, etc., and it carries out initiation, control, and diagnostic operations for the processor array and interconnection network. It is also responsible for communication between the processor array and peripheral devices.

The interconnection network is a 32-circuit crossbar switch with 32 input channels and 32 output channels. Each pair of input/output channels gives each processor bidirectional communication capabilities, so this interconnection network can be connected to a maximum of 32 processors. Moreover, programming software for controlling this interconnection network system can be used to form these processors into a parallel processing system with a variety of topological architectures such as an array type or tree structure, cube or hypercube structure, and so on.

The function of the root processor is to be responsible for advance processing of user jobs such as editing, compiling, connecting, and so on, and it carries out allocation, dispatching, and control of the other processors in the system. It is also the channel for data transmission between the processor array and the host machine. This point can be seen from the architecture. Moreover, the root processor also participates in parallel processing of jobs.

Each processor in the processor array (including the root processor) uses a 20 MHz T800 processor and has 2MB of local memory (LM). The function of this local memory is to access the program codes and data that must be executed by this processor system.

One can see from the architecture that the parallel processing system uses a loosely-coupled and point-to-point communication structure. The advantage of point-to-point communication is that it permits the addition of an unlimited number of processors without the possibility of problems appearing like those in a similar tightly-coupled and bus architecture. The basic

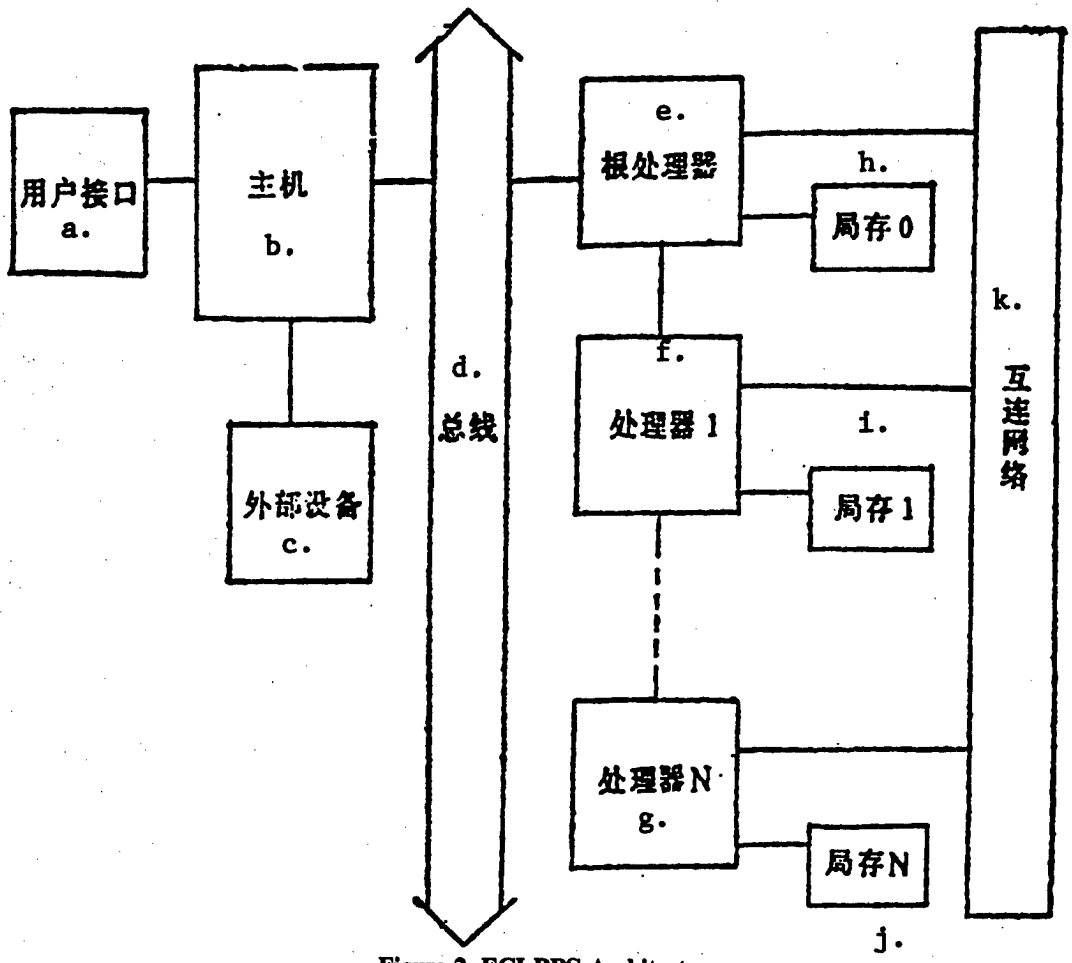


Figure 2. ECI-PPS Architecture

Key: a. User interface; b. Host computer; c. Peripheral devices; d. Bus; e. Root processor; f. Processor 1; g. Processor N; h. Local memory 0; i. Local memory 1; j. Local memory N; k. Interconnection network

architecture of the parallel processing system is a pipelining arrangement, but reflective alterations can be made via the interconnection network according to job processing requirements and architectures.

IV. Software Environment

Control of normal system operation involves the operation of system control modules. Before describing the working principles of the system control modules, we want to discuss the functions of the overall system control modules, the communication protocol for the parallel processing system and host computer, and the design of a device actuation program under XENIX.

A. Functional requirements

Besides the computing functions of the processing array itself, it must also control the required services provided by the software. These system services mainly involve system services, file processing, operating system control, and other functions.

1. System service functions: the requirements include system initiation and termination, providing a clock, environment changes, and so on.

2. File processing functions: the requirements including opening, closing, and reading and writing files, information transmission, etc.

3. Operating system control functions: the requirements include interrupt processing, system memory reading and writing, input/output port accessing, etc.

B. Communication protocol

In the overall system, there are two types of communication. One type is communication between the host machine and the processing array and the other is

communication between each of the processors in the processing array. Their formats and communication modes are different.

1. Communication between the host computer and the processing array

Functions to be executed — ! to. A1? from A2

to, A1 — Execution of instruction

from. A2 — Resulting value

In the format, ! is an instruction execution request received through the channel by the host machine from the processor array and then processed according to the corresponding requirement. ? is the result of the execution transmitted to the processor array through the channel after the host machine completes the instruction.

2. Communication among the various processors in the parallel processing array

Along with providing parallel languages, the INMOS Company also provides users with an intercommunication process among the processors. Its format is:

CHAN-IN(variable, input channel number)

CHAN-OUT(variable, input channel number)

CHAN-IN( )indicates that the variable is transmitted into the input channel

CHAN-OUT( )indicates the data received from the output channel.

There is mutual correspondence among them and communication only occurs when the input path and output path are in the same states. Otherwise, it remains in a wait state.

C. Design of parallel processing system hardware drive programs under a XENIX operating system environment

Because control and utilization of devices by the UNIX series of operating systems is different from the DOS operating system, it cannot directly use device ports in applications programs for reading and writing like DOS can. It requires that when an applications program uses devices, it must be carried out according to the file mode being utilized, and the drive program design must provide the applications program with several interfaces such as file modes, which mainly include opening, closing, reading and writing, other control, and so on, after which it restructures the operating system kernel. Implementation of this process as a whole is more difficult that under DOS. Below, we will only plot the working principles and design of a drive program under XENIX and will not provide additional descriptions under DOS.

1. Working principles of the drive program

As shown in Figure 3, when an applications program uses devices it can enter system calls (such as field=open ("device name", opening mode)) into the operating system kernel and the kernel then enters the corresponding drive program codes according to the parameter called. In this way, the drive codes control the devices according to requirements.

2. Drive program design

Based on the requirements of the operating system and the actual characteristics of the parallel processing system hardware, we designed a drive program for it that contains six processes. They are opening, closing, reading, writing, initialization, and communication processes. Opening and closing are the necessary preparatory work requested of the operating system for the reading, writing, and other processes. The reading and writing processes are simply data reception and transmission. These two processes also include detecting and processing device error states, while the initialization and communication visual applications programs use

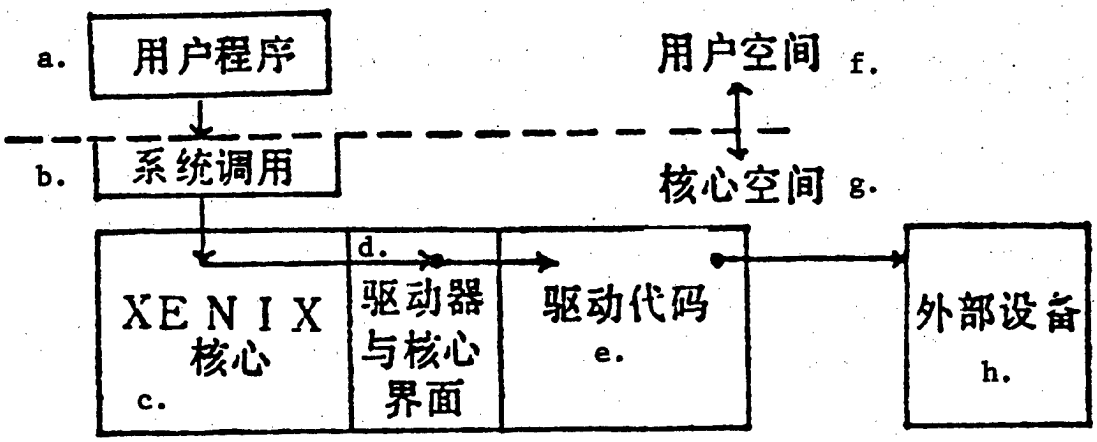


Figure 3. Working Principles of Drive Program Under a XENIX Environment

Key: a. User program; b. System calls; c. XENIX kernel; d. Driver and kernel interface; e. Drive codes; f. User space; g. Kernel space; h. Peripheral devices

the actual conditions of the device for selection and utilization, and they complete several initializations, port selections, and so on. These processes must use the special tools provided by the operating system to compile them and connect into the kernel.

D. Working principles of the system control module

The system control module of the parallel processing system serves as a sub- module of the operating system that runs together with the operating system in the host machine, coordinates and manages the overall parallel processing system, and facilitates transplantation of the system control module into other operating system environments (such as the DOS environment). It integrates processing array management, equipment request processing, communication coordination, management task dispatching, and other functions in this module, whereas the operating system manages system software resources (such as file systems, etc.) and hardware resources (such as standard input/output devices, etc.). This turns the most direct control and management tasks over to the system control module for completion. Below, we focus on a description of the flow process for the operation of this module (illustrated in Figure 4).

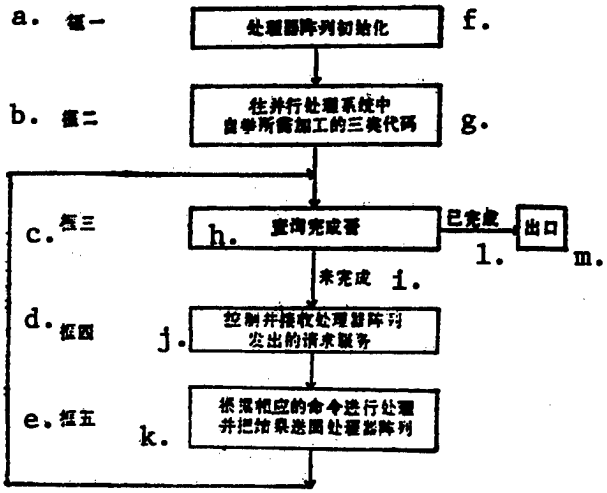


Figure 4. Parallel Processing System Control Module Flow Chart

Key: a. Frame 1; b. Frame 2; c. Frame 3; d. Frame 4; e. Frame 5; f. Processor array initialization; g. Self-selection of three categories of codes that require processing from the parallel processing system; h. Inquire if completed or not; i. Not completed; j. Control and receive service requests emitted by processing array; k. Carry out processing based on the corresponding instructions and transmit results to processor array; l. Completed; m. Output

Frame 1: The system control module presets the working state of the processor system and places it into a preparation state for receiving tasks such as processing array reset, error detection, etc.

Frame 2: The system control module transmits three types of codes (source codes, intermediate codes, object codes) to the processor array. These codes are generated, respectively, by the editor, the compiler on the root processor, and the chain linker.

Frames 3 and 4: The root processor is used to control the processor array (because all of the processors in the processor array must pass through the root processor before they can communicate with the host machine), inquire of its execution state, receive requests for data transmission, etc.

Frame 5: Processing of service requests from the processor array, such as file reading and writing and other related operating system system call requests, and feedback of processing results to the processor array.

V. Parallel Processing System Software and Hardware Characteristics

A. Primary characteristics of the software environment:

1. The ECI-PPS can run under both the DOS and XENIX environments, which has expanded its scope of suitability.

2. It provides serial C, FORTRAN, and PASCAL and parallel C, FORTRAN, OCCAM, and other language operating environments under DOS and XENIX.

3. Source programs under DOS can run without further modification under XENIX, and the same holds true for the opposite case. This makes program design very convenient.

B. Primary characteristics of the hardware system:

Modularized architecture, and each processor system in the ECI-PPS employs modular ideas in the design.

The system-level design leaves expansion and restructuring capabilities.

1. The changeable topological architecture and interconnection network software controllability permit corresponding changes to be made in this system according to different situations and different algorithm structures.

2. A loosely-coupled and point-to-point architecture. This can reduce the overhead arising from communication and competition.

3. RISC technology applications.

VI. Performance Evaluation

To make a quantitative description of the performance of the parallel processing system as a whole, we selected a testing program from the Gould Minisupercomputer Company and a large number of data computation programs, and ran user jobs on this system and ran identical jobs on other types of computers to compare and analyze the amount of time required.

A. List of testing programs

1. WHETSTONE: used to test floating point operations.

2. DHRYSTONE: used to test system input/output performance.

3. High-order matrix operations.

B. Performance evaluation

When conducting the performance evaluation, we mainly used an EC-386 developed by the East China Computing Technology Institute (25MHz clock, with CACHE) and representative computers from foreign countries as models for comparison with the parallel processing system. Tables 1, 2, and 3 list the results of the performance comparisons.

Table 1. WHETSTONE Program Test Table (units: K/S)

Type of computer and utilization environment	Single precision	Dual precision
EC-386 25 MHz XENIX	20	21
EC-386 25 MHz DOS	72	85
EC-386 25 MHz (+387) DOS	679	-
Processor array (One T800)	1587	1639
Processor array (Two T800)	2646	2703
Processor array (Four T800)	4831	4878

Table 2. DHRYSTONE Test Table (units: DK/S)

Type of computer	Master clock speed	Operating system	Performance
VAX-11/750	-	UNIX 4.2	877
IBM 4341	16.67 MHz	UTS 5.0	3685
SUN/75	16.67 MHz	SUN 4.2	3571
EC-386	25 MHz (CACHE)	XENIX V	6250
Parallel processing system (One T800)	20 MHz	TDS	4410
VAX 8600	-	UNIX 4.3	7088
VAX 8600	-	VMS	7142

Table 3. High-Order Matrix Sample Operation Comparison (Units: Seconds)

Number of cycles	10	50	100
EC-386 DOS	48	240	481
EC-386 XENIX	72	361	722
Processor array 1	6	28	55
Processor array 2	3	15	30
Processor array 3	2	12	23
Processor array 4	2	8	15

Note: The 1, 2, 3, and 4 after the processor arrays in Table 3 represent, respectively, one, two, three, and four T800 processors.

The ECI-PPS is a highly parallel processing system composed of four T800 chips (actually, it can be configured with a maximum of 32 processors). Interconnection among each of the processors is done via program control using C004 network interconnectors for direct connection among each of the processors. This interconnection arrangement enables relatively rapid information exchange among processors. The performance comparisons in Table 1 show that with a single T800, the operating speed ratio between this system and the EC-386 is 20:1 and that if the EC-386 is fitted with an 80387, their speed ratio is 2.4:1. When the system has two T800s, however, the speed ratio is 4:1. Because DHRYSTONE mainly concerns input/output management and system resource management in the parallel processing system (with only one T800 operating) is achieved via the system control module, processing by this system in this area is somewhat slower. When carrying out large-scale matrix operations, the parallel processing system has a higher throughput speed than the EC-386.

VII. Conclusion

Parallel processing technology was proposed quite some time ago, but it has only developed quickly after very large-scale integration provided a powerful foundation for this type of technology. The architecture of the chip computer series of chips developed by the INMOS

Company has unique functions. It melds the processor, memory, communication protocol, and peripheral device functions into an integrated whole. This increases the working capacity of each chip and they already have formed internally a set of timing and control signal sequences and communication chain circuit functions. They provide an excellent foundation for putting together mainframes and supercomputers with various types of topological architectures.

Zou Ling [6760 3781], Zhu Yuqing [2612 5148 3237], Wang Zhongkang [3769 0112 1660], Li Xingchuan [2621 5281 2504], and other comrades also worked on this project.

[*]Xu Yuhui was born in 1961 and graduated from the Computer Department at Shanghai Railroad College in 1982. He received a Master's degree from the Chinese Academy of Sciences in 1987. He currently works as an engineer. He is mainly involved in research on parallel processing, RISC technology, interconnection networks, and so on.

References

[1] INMOS Limited, "Transputer Development System", Prentice Hall, 1988.

[2] INMOS Limited, "Transputer Databook", Bath Press Ltd., Bath, 1988.

[3] INMOS, "Parallel C User Guide", 3L Ltd., 1988.

Multimicroprocessor System AP85

92FE0867C Beijing JISUANJI YANJIU YU FAZHAN [COMPUTER RESEARCH AND DEVELOPMENT] in Chinese Vol 29, No 7, Jul 92 pp 10-15

[Article by Zhou Yaorong [0719 5069 2837] and Zhang Daqing [1728 1129 1987] of the Ministry of Aerospace Industry Computing Technology Institute, Xi'an: "Multimicroprocessor System 85"; MS received Aug 88]

[Excerpts] **Abstract:** This article provides a detailed description of the architecture, working principles, and software and hardware composition of the AP85 multimicroprocessor system and integrates with the implementation technology of the AP85 system for a discussion of several problems common to multimicroprocessor systems. The article concludes with induction and summarization of the AP85 system's applications conditions and system characteristics, and it offers some proposals for further improvements in the system.

I. Introduction

Computers have undergone more than 40 years of development and their performance has been improved by many numerical grades. During the development process, facing restrictions by hard component technology and techniques on single processor speed, computer designers have proposed several technologies for increasing computing speed. The main ones are pipelining technology, vector flow and shared component technology, and parallel processing and RISC technology, which are still rapidly developing. It is not hard to discover that these technologies exploit and utilize latent parallelism at different levels, so an ideal computer system can be thought of as one that should be capable of exploiting parallelism at all processing levels to concentrate various types of technologies into one body to obtain ultrahigh computing performance. The computer architecture that approximates this ideal model should be led by multimicroprocessor systems.

Since the 1980's, the surging development of VLSI [very large scale integration] technology has led to the appearance of 32-bit high-performance microprocessors and a steep decline in hardware prices. These have greatly spurred R&D on multimicroprocessor systems. To date, there are at least several dozen types of multimicroprocessor systems in the United States, Japan, and countries of Western Europe. The better known ones include the iPSC-VX and Butterfly from the United States and the PAX-64 from Japan, all of which have several dozen processors installed. All types of facts have proven that research work on multimicroprocessor systems has now made the transition from the experimental research stage to the applied research stage and that research on multimicroprocessor systems has now become a topic with important development directions.

In the early 1980's, the Ministry of Aerospace Industry Computing Technology Institute in Xi'an began doing research on multimicroprocessor systems and felt that a system composed of several low-cost microprocessors was an effective way to implement a high-performance computer capable of meeting the urgent requirements of many departments within China for high-speed computers, especially aviation departments. In 1985 we began development of the AP85 multimicroprocessor system with the anticipated early-90s objective of developing the AP256 supercomputer with a peak speed at the 100 million instructions per second [MIPS] grade.

II. AP85 Architecture

The AP85 is an experimental loosely-coupled multimicroprocessor system. Its goal is to support exploration and research on multiprocessor architectures, parallel algorithms, and multiprocessor support software. The overall system is composed of a host computer, a communication controller, and an array composed of 16 asynchronous microprocessors. The host machine and communication controller are connected via a high-speed bus with each microprocessor in the array, which is also called array element processor interconnection. Each array element processor is also directly interconnected via a point-to-point channel with its four adjacent array element processors, forming an FNN (four nearby node interconnection) (see Figure 1).

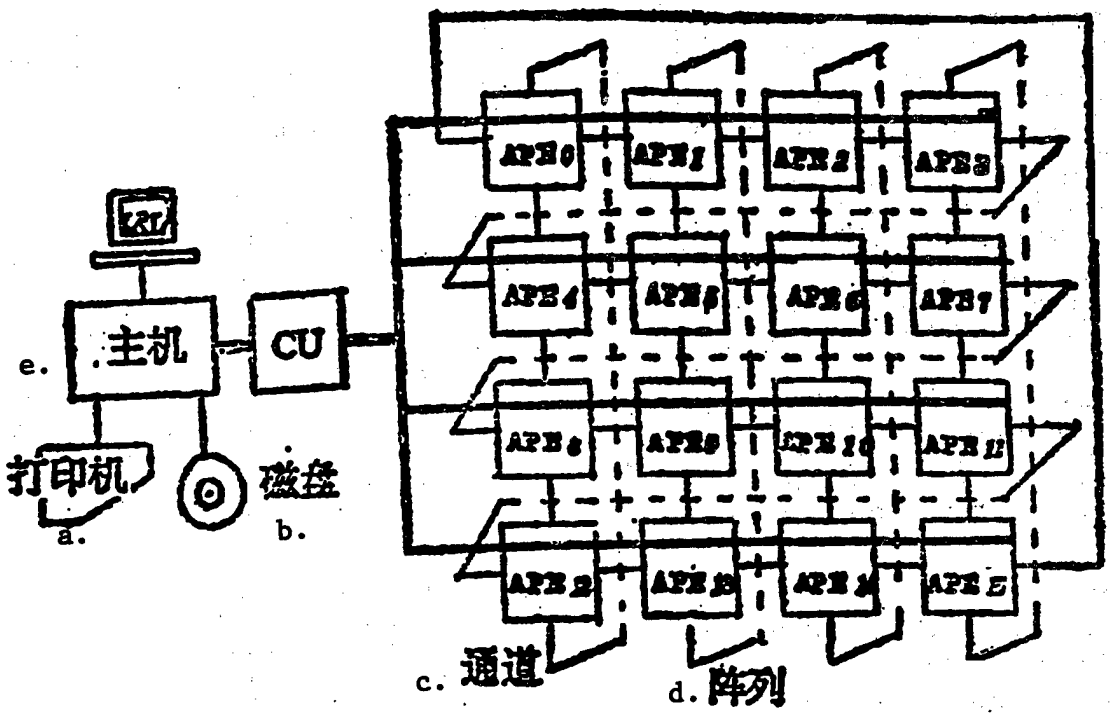


Figure 1. AP85 Architecture

Key: a. Printer; b. Disk; c. Channels; d. Array; e. Host computer

The AP85 microprocessor array makes use of the enormous flexibility of FNN two-dimensional network interconnection. This interconnection is suitable for direct mapping of one-dimensional, two-dimensional, ring-shaped, tree-shaped, multidimensional, and other mathematical models. For a multiprocessor system composed of P nodes and using a certain number of ring-shaped connections, the average path of the network is P/2. If converted to two-dimensional network interconnection, the average path is $P^{1/2}$. The average path of three-dimensional network interconnection is $3P^{1/3}/2$. Thus, for a relatively large-scale system composed of 1,000 nodes, the ratio of the average paths for one, two, and three-dimensional interconnection is 512:32:15. Obviously, two-dimensional interconnection is far superior to one-dimensional interconnection, whereas the superiority of three-dimensional interconnection relative to two-dimensional interconnection is much less. Taking into consideration the cost and degree of difficulty involved in interconnection, we feel that two-dimensional interconnection is a more ideal interconnection mode for medium and small-scale systems.

The AP85 utilizes in-task process one-level parallelism. It is different from multiple component processors or vector processors it that it uses local instruction-level parallelism, which avoids restriction by scalar operations and the need for high-speed elements and complex logic structures. Compared to data pipelining processors, the AP85 uses a moderate-grained data pipelining structure. Although the degree of parallelism it utilizes is not as high as data pipelining processors, its simplicity of control is appropriate for present technical levels.

The AP85 system uses a decentralized memory architecture and has no shared memory. Process synchronization is achieved automatically by the producer process transmitting data to the consumer process, which eliminates memory conflicts, complex switching network control, and a series of other problems related to a shared memory architecture. Although for a rather small scale system a multiprocessor system with a distributed memory architecture can result in substantial losses in overall system performance arising from the relatively large communication overhead, in a multiprocessor system with superhigh performance that is composed of tens of thousands of nodes the adoption of a decentralized memory architecture would appear to be unavoidable. [passage omitted]

IV. System Hardware Description

A. Composition of microprocessor array

The microprocessor array in the AP85 system is composed of 16 completely identical array element microprocessors. Each array element microprocessor is an integral microprocessor system composed of the following parts.

1. Intel 8086 microprocessing element and Intel 8087 coprocessor

The Intel 8086 is a 16-bit microprocessor with a 20-bit address line that gives it a 1MB addressing capability. Its advanced internal architecture enables it to carry out 16-bit fixed point number arithmetic and logic operations. The Intel 8087 is a coprocessor used for the special purpose of processing numerical data operations that is capable of directly carrying out 32-bit, 64-bit, and 80-bit floating point operations and 18-bit BCD data operations. Its floating point format is completely identical to the IEEE-751 floating point standards used in standard FORTRAN. At a 5 MHz working frequency, a system composed of an Intel 8086 + 8087 has a floating point operation capability of up to 0.5 MFLOPS.

2. Array element microprocessor memory

Each array element microprocessor has its own local memory that includes 128K of RAM and 48K of ROM. The ROM is used to access the array element microprocessor support software. With the exception of part of the RAM being allocated for system use, most of it is space available for use by users. The RAM is a dual-port memory that is both the main memory for the array element microprocessors and capable of host machine reading and writing. The memory capacity of the array element microprocessors as a whole can be expanded to 1MB.

3. Intel 8255 parallel interface controller and Intel 8251 serial interface controller

The Intel 8255 parallel interface controller is used to control communication among adjacent array element microprocessors. Each array element microprocessor is configured with four channels that are connected, respectively, to the four surrounding array element microprocessors. The serial interface controller is mainly used to prepare for external connection to terminals to facilitate debugging, and it can be connected to external input/output devices.

4. Other control components and logic

Besides the components described above, the array element microprocessors also have interrupt controllers, timing timers, clock generators, RAM dual-port control circuits, and other components and logic-aided construct systems to ensure control of the array element microprocessors by the host computer and carry out communication functions between the host machine and the array element microprocessors and among the array element microprocessors.

B. Composition of the host computer

The host computer in the AP85 system is made by expanding an IBM PC-XT microcomputer. The expanded portion includes an array drive board and an interrupt control board. The function of the array drive board is to provide sufficient drive current for normal operation of the array microprocessors while the function of the interrupt control board is to load and mask the interrupt signals sent by the microprocessor array to the host machine to assist the host machine in managing implementation of the microprocessor array.

C. Communication controller

The communication controller CU is the hub linking the host computer and the microprocessor array. It is composed of an Intel 8089 I/O processor with added control logic. The Intel 8089 is a coprocessor used especially for input/output processing. It is connected in a remote fashion to the host computer Intel 8088 CPU in this system. It treats the array microprocessors as peripherals and utilizes its high-speed DMA transmission functions to carry out data transmission from array element microprocessor to array element microprocessor, from array element microprocessors to the host computer, and from the host computer to the array element microprocessors. The broadcast transmission functions from the host computer to the microprocessor array and from the array element microprocessors to each array element microprocessor gives this system an extremely high transmission efficiency. The Intel 8089 can achieve arbitrary connection of 8-bit or 16-bit peripherals with 8-bit or 16-bit processor busses. This permits connection of the Intel 8088 which has an external data bus width of 8 bits with the Intel 8086 which has a data bus width of 16 bits.

D. Interconnection channels

1. Local interconnection channels. Each array element microprocessor has four parallel 8-bit input ports and four parallel 8-bit output ports that form dedicated input/output channels with the four adjacent array element microprocessors. The local interconnection channels are controlled only by the array element microprocessors and are used for random data communication with the adjacent array element microprocessors.

2. Global system bus: This is a 16-bit parallel time-division multiplexing bus that can provide the array element microprocessors in the array with point-to-point communication and broadcast communication. The global system bus connects the host computer to all the array element microprocessors. The host machine can use the global bus to load programs and data into the array element microprocessors, and it can extract the results from the array element microprocessors.

V. System Software Architecture

Figure 2 shows the model of levels in the AP85 system software architecture. The left half is the program running in the host computer. It is responsible for global control and provides support for operation of the system as a whole. The right half is the program running in the array element microprocessors. It is responsible for task distribution and processing. The multimicroprocessor software added to the PC-DOS operating system includes:

- **Array control software:** This is the interface software for the host computer and microprocessor array. It is composed of physical instructions for completing

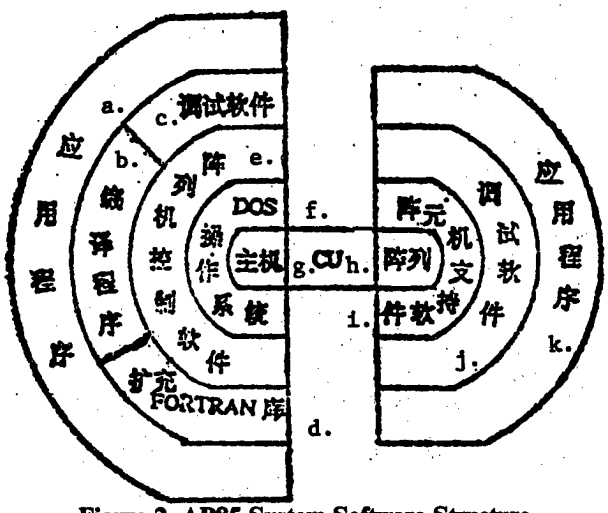


Figure 2. AP85 System Software Structure

Key: a. Applications programs; b. Compiling programs; c. Debugging software; d. Expanded FORTRAN library; e. Array processor control software; f. DOS operating system; g. Host computer; h. Array; i. Array processor support software; j. Debugging software; k. Applications programs

structure startup, reset, and synchronization and for controlling bus communication.

- **Debugging software:** In a multiprocessor environment, it can support the operation of static and dynamic debugging DEBUG programs in parallel programs. It includes an assembly language level-one debugging program MDEBUG and a FORTRAN source program level-one debugging program FDEBUG.
- **High-level language support software:** This is achieved by using the serial FORTRAN source language as a foundation and adding parallel statement functions. With a prerequisite of not changing the FORTRAN compiling program, it can convert the source program codes written by the FORTRAN language document into object codes for execution in a multiprocessor environment.
- **Array element microprocessor support software:** This is the environment software that remains in the array element microprocessors to support effective operation of user programs. Its primary components are the array element microprocessor initialization program, self-checking program, communication program, and a series of software and hardware interrupt service programs.
- **Array element microprocessor debugging programs:** These are the debugging programs inside the host computer that were designed to do dynamic debugging of programs running in the array element microprocessors. Their functions include setting up breakpoints, single-step tracking, I/O port reading and writing, and so on. They are automatically loaded into each of the array element microprocessors during program debugging.
- **Applications programs:** These include a linear equation group solution program, partial differential equation solution program, finite element computation program, multiple integral solution program, hydrologic processing and seismic processing computation examples, and other programs.

VI. Applications

With the objective of testing system performance and studying parallel algorithms, we selected several representative and general problems in engineering computing for computation and solution in the AP85 system. Existing typical computation examples including using the SOR method to solve partial differential equation category-one boundary value problems, linear equation solutions, multiple integral solutions, matrix operations, finite element computations, and so on.

The results of the computation tests indicate that in solving problems with a relatively large number of operations and relatively small communication overhead, the AP85 can provide relatively ideal speedup and efficiency. For example, the SOR method was used to solve partial differential equation category-one boundary value problems and provide linear equation group loose iteration method solutions, and we used the Monte Carlo method and number theory network method for multiple integral solutions, overall program computations in finite element computations, matrix multiplication, and other computations using 16 array elements, and the system speedup ratio was greater than 10 times.

For the matrix addition method, however, because the number of array element microprocessor computations was too small and not large enough in proportion to the amount of data transmission, the speedup ratio was only about 3 times and the improvement in speed was not significant enough. In using the G-J method to solve linear equation groups, the speedup ratio was also not very large in the system at the present scale, only 3 times-plus. This shows that certain problems themselves not suitable for parallel solution or the improper selection of algorithms can result in a situation in which the system solution rate is not high.

The AP85 system displayed excellent performance when solving several real engineering problems of users. When solving optimum parameter selection problems for the hydrological forecasting model at the Xi'an Central Hydrology Station, the system speedup ratio was 14.37. Similar methods were used to solve seismic wave synthesis problems in engineering seismology in the PDP- 11/23 and AP85 system, and we discovered that the solution speed for this problem in the AP85 system was 16.7 times the speed of the solution of the same problem in the PDP-11/23.

VII. Conclusion

The AP85 system was successfully developed in April 1987 and passed ministry- level examination and acceptance in August 1987. The AP85 is a relatively general-purpose prototype that provides a dependable environment for conducting research on multiprocessor architecture, parallel algorithms, and multiprocessor system support software, and it has laid a foundation for further development of even larger scale or special-purpose multiprocessor systems.

In all, the AP85 system has the following characteristics:

1. It employs a relatively ideal two-dimensional FNN interconnection network that facilitates direct mapping of algorithms for large numbers of applications problems into the array network.

2. The communication modes are relatively flexible and it has two communication modes, high-speed DMA global bus communication and parallel node cross channel communication.

3. The host computer and array element microprocessors adopt a similar series of processors and components that facilitate the use of the host machine software resources.

4. It has excellent expandability that facilitates the addition of nodes to construct larger scale systems. Added to the fact that the components and host machine it uses are the most popular in the market, this makes it easy to use higher grade compatible components to raise the system performance grade.

5. There is relatively abundant software support for the system and it can support the use of high-level language FORTRAN programming.

6. It is configured with the relatively powerful assembly language level-one and high-level language level-one debugging tools MDEBUG and FDEBUG. They include a FORTRAN source language level-one debugging tool, which is an entirely new software development tool for IBM-PC/XT users.

The AP85 system is an experimental multimicroprocessor system and it still has problems like less-than-powerful software functions, rather low system performance, and so on. To improve the overall performance of the system by numerical grades, we are preparing to focus on development work in the following areas.

1. Configuration of a parallel compiling and more perfect function multiprocessor operating system so that normal users can use the system without too much difficulty.

2. Adoption of an Intel 80386+80387 high-speed processor and special vector and floating point operations units for the processing units of the array element microprocessors and simultaneous increases in the system scale.

3. Adoption of a higher speed bus and cross channels in conjunction with configuring each array element microprocessor with special processing elements to manage communication and reduce the overhead taken up by communication.

4. Further development of research work on parallel algorithms and the establishment of an intersecting parallel programming environment for users to aid users in writing high efficiency parallel programs.

Thanks: The authors offer their sincere gratitude to all members of the AP85 topical group, especially to Wei Shaoxian [7614 4801 6343], Peng Dewen [1756 1795 2429], Xie Peng [6200 3403], Wang Ji [3769 3444], Liu Xiaomin [0491 1420 3046], Wang Yongbao [3769 3057 1405], Feng Guangmei [7458 0342 2734], Han Wei [7281 3555], Li Rui'e [2621 3843 1230], and other comrades. Xi'an Jiaotong University professor Zheng Shouqi [6774 1343 3825], Northwest Polytechnical University professors Kang Jichang [1660 4949 2490] and Han Beixuan [7281 0554 6513], and Northwest University professor Hao Kegang [6787 0344 0474] offered warm support to this topic, and we would like to express our gratitude.

(references omitted)

Software Pipelining Based VLIW Architecture

92FE0867D Beijing JISUANJI XUEBAO [CHINESE JOURNAL OF COMPUTERS] in Chinese
Vol 15, No 7, Jul 92 pp 481-490

[Article by Su Bogong [5685 0130 3797], Tang Zhizhong [3282 1807 1813], Zhao Wei [6392 1550], and Wang Jian [3769 0494] of the Qinghua University Department of Computer Science and Technology, Beijing*: "VLIW Architecture Based on Software Pipelining Technology"; MS received 24 Aug 90]

[Excerpts] **Abstract:** This paper introduces a VLIW [very long instruction word] multiprocessing element single-chip computer now being developed. The architecture of this machine is based on URPR [unrolling, pipelining, rerolling] software pipelining technology and uses a pipeline register file to reduce the interbody dependent distance, which enables full exploitation of fine-grained parallelism and thereby enhances loop body overlapping and greatly shortens the length of the loop body after optimization. The results of simulation experiments indicate that this architecture can attain high performance when matched with an optimizing compiler.

I. Introduction

[passage omitted]

Our basic idea was to use current VLSI [very large scale integration] technology in an effort to integrate several processing elements on a single chip to construct a signal processor with a VLIW architecture and to rely on an optimizing compiler to fully exploit instruction-level fine-grained parallelism, thereby greatly improving its

performance compared to single processors under identical technical conditions. In addition, the description provided below shows that while the adoption of a VLIW architecture overall requires the use of more transistors, the overall structure is relatively integral and simple, so it is not hard to implement.

Because the optimizing compiler has an extremely great effect on the performance of VLIW architecture, all types of VLIW architectures are actually based on certain key optimization algorithms. For example, the TRACE series from the MULTIFLOW Company is based on a path scheduling algorithm. The VLIW architecture we are proposing is based on a URPR software pipelining loop code optimization algorithm, so we call it the URPR-1 machine. We proposed the URPR algorithm in 1986 as being very suitable for loop code optimization in signal processing and image processing, especially because it has relatively good time benefits as well as relatively good space benefits. This point is especially important with single-chip processors that have a limited amount of storage.

II. Using Hardware Support To Improve the Benefits of the URPR Algorithm

In signal processing, image processing, and other program codes, the execution time for loops, especially inner-level loops, accounts for a very large proportion. Thus, loop optimization is the key to improving program execution efficiency and overall system performance. Software pipelining is an effective technique for loop optimization. With a prerequisite of not changing the program semantics, iteration is carried out for loop bodies at different levels to fully exploit the parallelism of hardware resources and thereby shorten loop execution times.

We proposed the URPR software pipelining algorithm in 1986[9,11]. It has the advantages of a low degree of computing complexity and good time and space benefits.

Its main principles are that the loop bodies are first opened up to K in number and these K loop bodies are pipelined and loaded, after which they are drawn in to obtain a new optimized loop body whose minimum length is the interbody dependent distance D of the original loop body. For this reason, two types of hardware support are required to obtain the optimum benefits:

1. Increasing the number of data paths and functional units to reduce resource conflicts.

2. In a situation of infinite resources, the minimum length of a loop body after optimization is equal to the loop body interbody dependent distance D, so hardware support is required to reduce D. We pointed out in reference [7] that reducing the register holding time can reduce D. Lam adopted a variable modulo expansion method in the WARP machine[6] in which the same variable located in

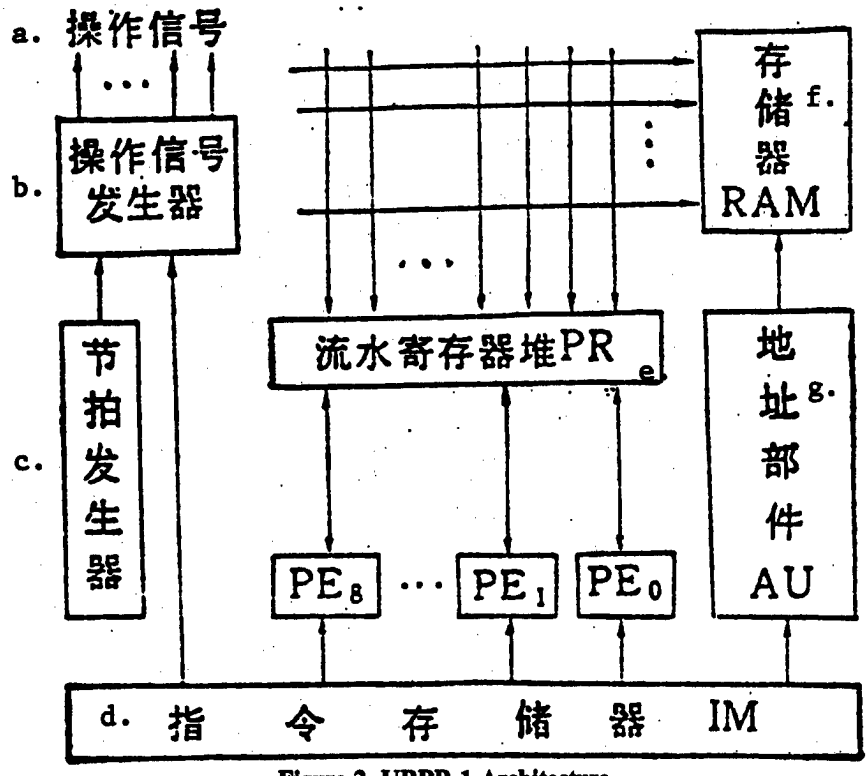


Figure 2. URPR-1 Architecture

Key: a. Operating signals; b. Operating signal generator; c. Beat generator; d. Instruction memory IM; e. Pipeline register file PR; f. Memory RAM; g. Address unit AU

different loops is given to different registers to increase the degree of interbody iteration. Here, we are proposing a new hardware support method for reducing D: the pipeline register file. [passage omitted]

III. URPR-1 Architecture

The URPR-1 is a fixed point 16-bit single-chip signal processor. Its architecture is illustrated in Figure 2. It is composed mainly of nine PEs [processing elements] with identical structures, the pipeline register file PR, the instruction memory IM, the data memory RAM, the address unit AU, the related control units, and so on. The operating cycle of the chip is 50ns.

Processors with this type of structure have the following characteristics:

1. High degree of parallelism in operation. The overall processor is capable of completing over 100 operations in one machine cycle. The operations that each processor can execute in parallel are: one multiplication operation, one arithmetic logic operation, eight inter-register data transmission operations, and two memory read-write operations.

2. The pipeline register is the core. The pipeline register can exchange data with the memory and it can transfer data between the register of its own PE and the corresponding register of an adjacent PE, which greatly improves the parallelism of operations.

3. Shared main memory. The data registers in the chip are shared by all the PEs. This structure can simplify program design and reduce the amount of communication among PEs, but hardware implementation is relatively complicated.

4. Good expandability. The three main parts—the pipeline register, instruction memory, and PEs—establish corresponding relationships similar to a bit slice structure. At present it has 9 PEs. If the number of PEs is increased, increasing the three parts simultaneously is all that is needed.

A. PE structure

Each PE is composed mainly of a rapid multiplier MUL, an arithmetic logic unit ALU, 16 registers, a certain number of multiplex switches, and so on, as illustrated in Figure 3.

The required number of operations for the ALU and MUL come from the pipeline register file and the operations results are sent back to the register file. The operations units do not have a direct relationship with memory. This is the architectural design idea of having the registers as the core, and this type of structure can increase the speed of data flow in the operations units and fully foster the efficiency of the rapid operations units.

To make an immediate increase in the number of operations required by the operations units, we adopted two measures. One was establishing several data paths between the registers and the memory and between the PEs and the corresponding registers. Now, each PE has a total of 18 data paths to the outside. The second measure

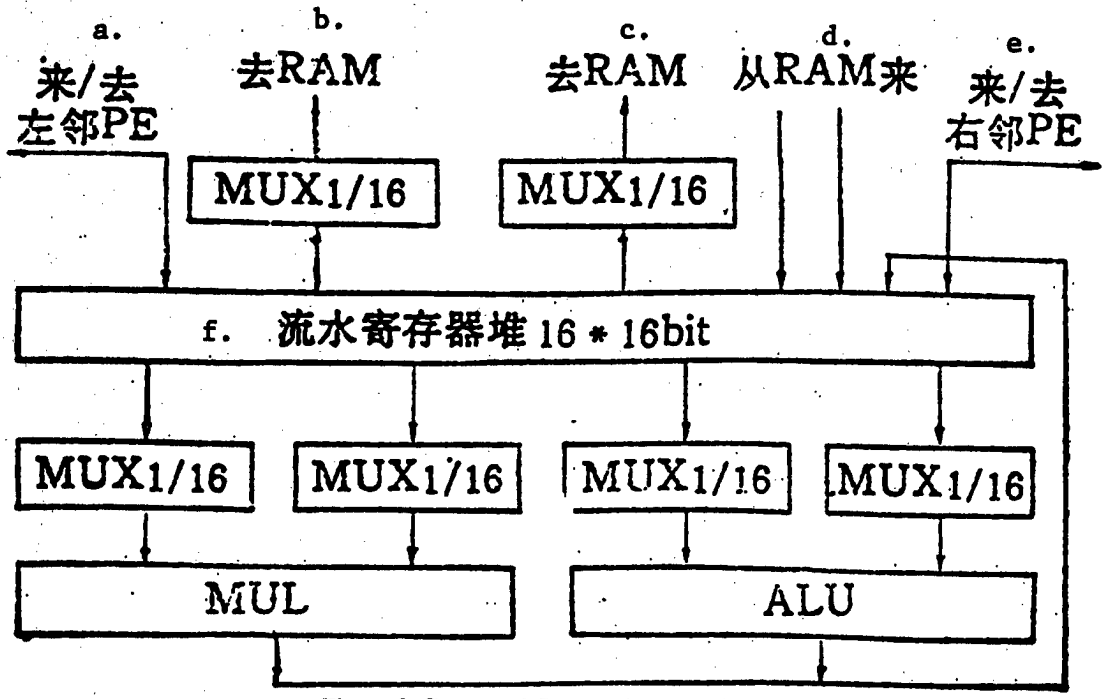


Figure 3. Structural Block Diagram of One PE

Key: a. From/to adjacent PE at left; b. To RAM; c. To RAM; d. From RAM; e. From/to adjacent PE at right; f. Pipeline register file 16 * 16bit

was to make data transmission from the registers to the outside and operation of the operations units operate in parallel and use compiler scheduling to ensure that the required number of operations enter the pipeline register file in advance.

The rapid multiplier completes a 16-bit fixed point multiplication in one machine cycle.

The rapid multiplier is used to achieve division. The completion of one rapid division requires three multiplications and three ALU operations.

The main ALU operations are logic operations, arithmetic operations, shift operations, and so on. Among them, three address forms are used for logic operations and arithmetic operations: two source addresses and one target address. Besides using one source address and one target address, shift operations also use a 4-bit expression shift bit number of another source address.

B. Pipeline register file

The 16 registers in each PE are divided into three categories:

1. Local registers, a total of eight, that are used for storing only those constants and variables that are read and written in their own PE. The function of these registers is identical to general purpose registers in traditional computers.

2. Leftward pipeline registers, a total of six. These registers have two uses. One is vertical pipelining to establish a data path between two arbitrary adjacent vertical registers. The use of vertical pipelining can directly achieve assignment operations without going through the ALU. The second is horizontal pipelining, flowing leftward toward the corresponding register on the adjacent PE. These types of pipelining can be used to achieve inter-segment variable transmission, meaning that variables are read out and written in in the same loop body.

3. Rightward pipeline registers, a total of two. Besides having vertical pipelining functions, these registers can also flow rightward to the corresponding adjacent registers. The use of this type of horizontal pipelining permits the transmission of inter-segment recursive variables, meaning variables that are written in in the same loop body but only read out during the next or several later loop bodies.

All of these three types of registers are combined to construct a pipeline register file. Figure 4 illustrates the connection relationship for eight of the pipeline registers.

Data can be transmitted via two methods among the registers inside each PE:

1. Via vertical pipelining operations among adjacent pipeline registers.

2. Via the ALU.

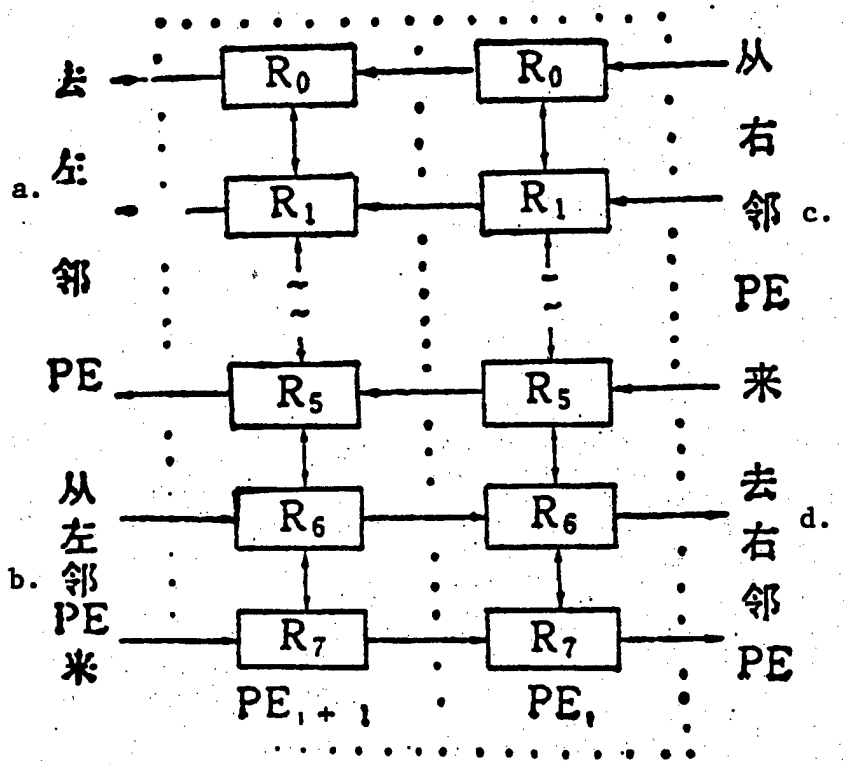


Figure 4. Explanation of Pipeline Register Connection Relationships

Key: a. To adjacent PE at left; b. From adjecent PE at left; c. To adjacent PE at right; d. From adjacent PE at right

The compiler ensures that every effort is made to use the first method by allocating the two variables to be transferred into the two adjacent registers, and it uses the second method only when necessary.

There are also two methods for data transmission between the registers inside each PE and the outside:

1. There are direct data paths between the pipeline registers and the registers in adjacent PE with the same number.

2. In one machine cycle, each PE can exchange two pieces of data with the memory in the chip.

The latter method is mainly used to access the original data required in the operation and save intermediate results. Data transmission between PE should make every effort to use the first method and the second method is only used when necessary.

C. Instruction set

Because the URPR-1 is oriented toward signal processing and image processing, we collected several dozen types of common signal processing and image processing algorithms and used the URPR software pipelining algorithm and optimizing compiler technology to carry out manual programming and optimization analysis of these algorithms. We also took into consideration the development situation for VLSI technology and realistic possibilities to design the instruction set described below.

The broad instructions for the URPR-1 have a total of 558 bits and are composed of three parts, as illustrated in Figure 5(a).

Each part is defined as follows:

1. Each PE field has 45 bits in a format as illustrated in Figure 5(b). In it, E is the multiplication enable bit. When E is set, $(SR_2)*(SR_1)\rightarrow DR$ is executed. Otherwise, there is no operation. OP is the ALU operation code that includes arithmetic logic operations, shift operations, and two types of multiplication support operations. $R_0,...,R_7$ are the register pipelining control fields and each register uses 2 bits for control.

2. Each memory read/write field has 14 bits in a format as illustrated in Figure 5(c). In it, the four types of codes for F represent, respectively, no read/write, memory write, memory read, and memory read based on FFT butterfly address conversion. The PE# and REG# operate together to point to a register in the register file and AC# is the number of the address counter. See Section D below for the address formation process.

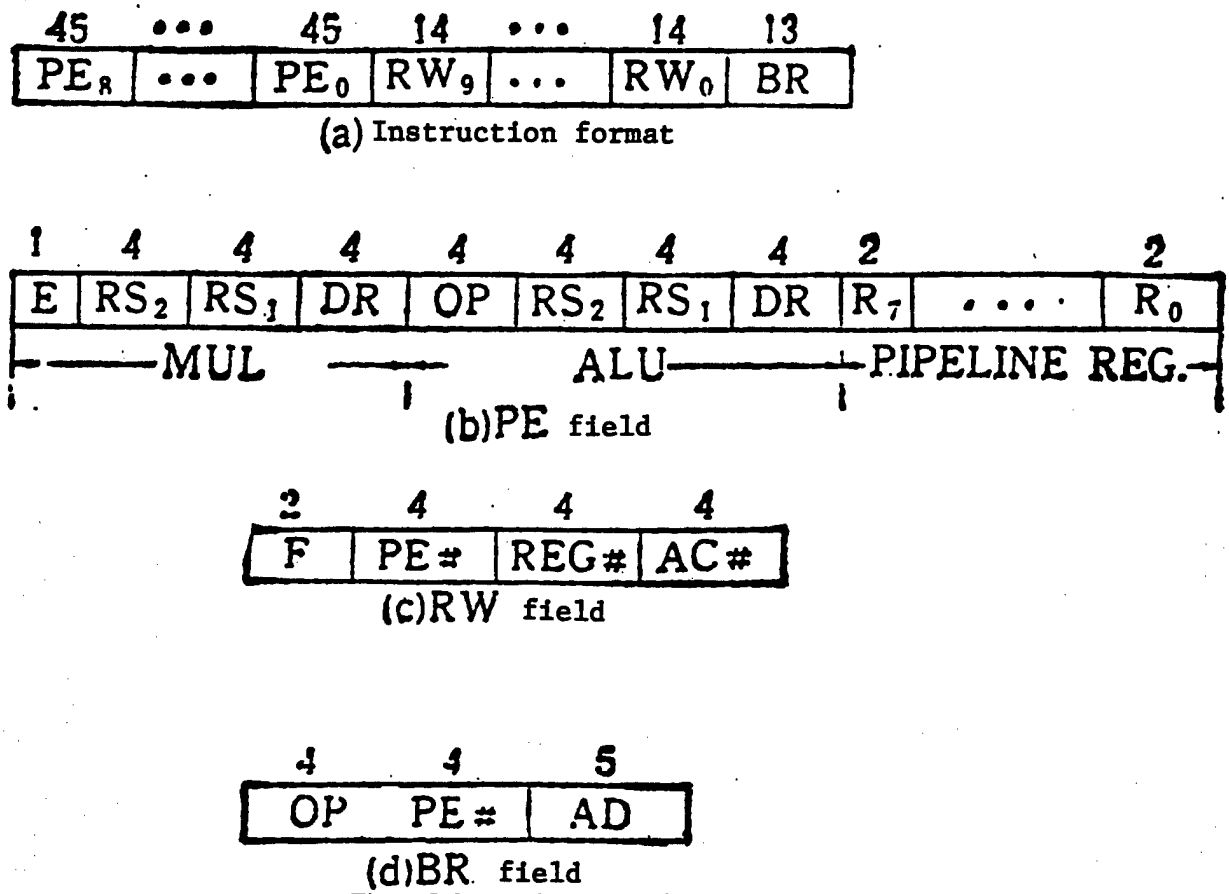


Figure 5. Instruction Format for URPR-1 Machine

3. The branch control field BR. We have stipulated that only one PE is permitted to execute a branch operation in one machine cycle. The format of the field is illustrated in Figure 5(d). In it, OP is the operation code, and its branch conditions come from the ALU operations result of that PE that is pointed out by the PE# field. AD is the branch target address, and it points to an instruction in the instruction memory IM. In addition, to match up with the conditional branch statements in high-level languages, we did not establish a condition code register. The branch conditions are obtained directly from the results of execution of the instruction.

D. Address unit AU

The five memories operating in parallel in the chip can read and write 10 pieces of data in one machine cycle that are controlled, respectively, by the 10 read/write fields in the instruction. Each PE is permitted to access the memory two times in one machine cycle but the total number of accesses by the nine PEs cannot exceed 10 times. Thus, the five memories and nine PEs actually constitute a 5*18*16 multiplex cross switch, or they may be called five memory busses, as illustrated in Figure 6. In the figure, PR_0,...,PR_8 represent, respectively, the pipeline registers in the nine PEs. The data busses for the memories are connected to each of the pipeline registers via a multiplex switch, latch, and three-state gate. AG is the address generator that is used to form the address for memory read/write. Its primary function is to achieve FFT butterfly address conversions. Moreover, it also includes five address registers and five 16-to-1 multiplex switches, and so on. The five address generators are used at different times and provide 10 addresses in one machine cycle. Address computation and memory read/write operations work in parallel in a pipeline mode.

The address counter is only loaded once at the time of machine initialization. During the program running process, a 1 is added to all 16 address counters when each loop is finished.

IV. An Example of an FFT Butterfly Operation

We will use a base-2 complex 1024 point FFT innermost layer loop butterfly operation as an example of URPR-1 system operation. [passage omitted]

Figures 7(c) and (d) [not reproduced] show that when the URPR-1 machine is computing an FFT innermost layer loop, each machine cycle carries out one butterfly operation. Completion of the entire FFT computation requires 10*512 butterfly computations, so the time required is: 50ns*10*512 = 0.256ms. Adding the pipeline filling and emptying time, the URPR-1 can complete base-2 complex 1024 point FFT computations in 0.26 ms.

V. Discussion

Table 1 compares the URPR-1 with the TRACE and WARP VLIW architecture machines in the two areas of architecture and program design.

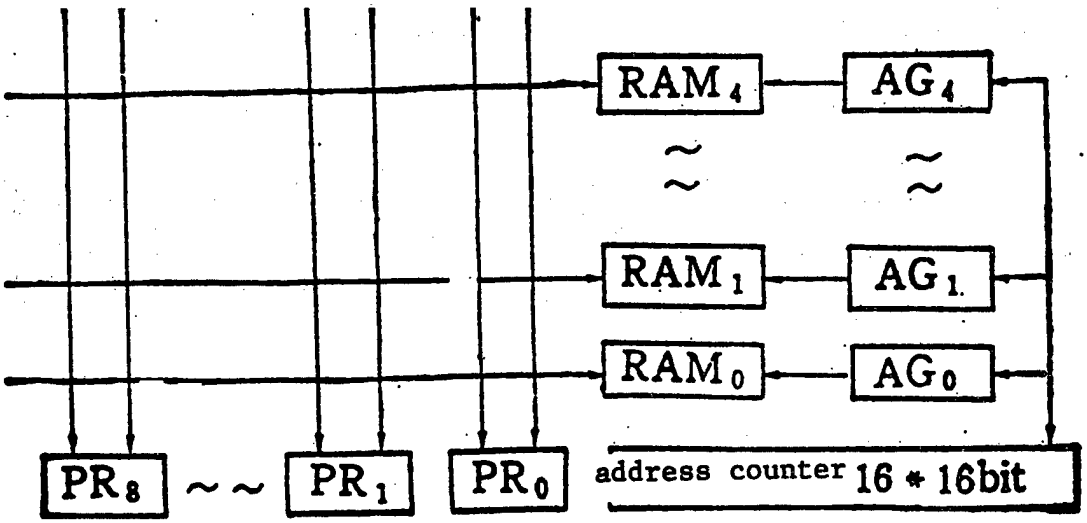


Figure 6. Composition of Address Unit AU

Table 1. Comparison of Three Types of Machines

Machine	Architecture	Program design
URPR-1	16-bit fixed point VLIW machine using a shared internal memory multiple PE structure, data transmitted between PEs via a pipeline register file, data transmission speed 160MWords/s.	Users use the C language for programming and employ an optimizing compiler with two-level software pipelining technology to develop instruction-level fine-grained parallelism, making full use of all of the machine's PEs and all functional units within each PE.
TRACE	A large VLIW computer configured with several integer processing elements and floating point processing elements, with busses interconnecting each of the processing elements with the others and the processing elements with the internal memory, each functional unit utilizes a pipelining structure.	Users can use the C or FORTRAN languages for programming and use a multiplex scheduling algorithm compiler to develop instruction-level fine- grained parallelism.
WARP	An MIMD structure composed of 10 cells, adjacent cells are interconnected via two data paths and one address path, each cell has a VLIW structure, with rather large local memory capacity and using a pipelining structure 32-bit floating point adder and multiplier.	Users use W2 language for programming, the compiler uses software pipelining technology to develop fine-grained parallelism for the program in each cell, coarse-grained parallelism among the cells is developed by users or by other compilers.

Comparing the URPR-1 and the TRACE, the TRACE is a superminicomputer oriented toward scientific computing that is configured with several integer processing and floating point processing elements, and the elements are interconnected via busses. The URPR-1 is a VLIW single-chip machine oriented toward signal processing and is only configured with fixed point multiplication and fixed point addition functional units. Adjacent PEs transmit data via pipeline registers. Compared to the bus arrangement, it has a broad bandwidth and is more appropriate for use in VLSI implementation. The optimizing compiler in the URPR-1 uses two-level software pipelining technology for loop body optimization, which is more effective that the route scheduling method employed in the TRACE compiler.

Comparing the URPR-1 and the WARP, the WARP is composed of 10 elements with more powerful functions and there is no shared memory for each element. Adjacent elements are configured with two data paths and one address path. In the URPR-1, however, adjacent PEs can transmit a maximum of eight pieces of data simultaneously via the pipeline registers, which enables full exploitation of fine-grained parallelism. In the area of program design, because there is coarse-grained parallelism among the elements in the WARP, partitioning must be done by the programmer, which makes user programming more difficult. Moreover, WARP requires users to use the W2 language for programming, so users cannot directly utilize existing applications programs in the signal processing and image processing realm. URPR-1 users do not have to partition the programs and can use the C language for programming, which aids in transplanting existing applications programs.

Comparing the URPR-1 with pipelining processors, the URPR-1 does not link all of its functional units into a hardware controlled pipeline structure. Instead, applications software pipelining technology partitions loop programs into each of the functional units. During the process of program execution, each functional unit seems to be linked into a software controlled dynamic pipeline, which gives it greater flexibility than pipeline processors so that it can be adapted to all types of applications programs and more effectively solve cutouts and other problems encountered in pipeline processors.

VI. Conclusion

The URPR-1 is a single-chip signal processor that uses a VLIW architecture. It has a 16-bit fixed point word length, an instruction word width of 558 bits, 2K of RAM and 2K of ROM on the chip, and about 600,000 transistors and 128 pins on the chip. Most of the chip is taken up by memory, registers, and non-complicated PEs.

At an operating cycle of 50ns, the peak computation rate is 360 MIPS and the signal transmission rate between PEs is 160 MWords/second.

The URPR-1 is configured with an optimizing compiler (see the first page of the next article). This compiler can convert applications programs written in the C language into machine codes and fully exploit instruction-level fine- grained parallelism. The prototype for the compiler has now been completed and the hardware design is in the simulation experiment stage.

Table 2. Simulation Results

Complex 1024 point FFT (base 2)	0.26 ms
FIR [finite impulse response] filtering (one accumulation)	6ns
IIR [infinite impulse response] filtering (eight coefficients)	100ns
Lattice filtering	50ns
Vector point accumulation (each element)	12.5ns

The results of simulation experiments for typical signal processing programs are given in Table 2. We expect broad application of the URPR-1 in signal processing, image processing, and other applications realms.

We offer our sincere thanks to professor Yue Zhenwu [1471 7201 0063] of the Qinghua University Microelectronics Institute for his useful assistance.

*This project was funded by the National Natural Science Fund. Su Bogong is a professor who is involved in research in the area of computer architecture. Tang Zhizhong is an associate professor who is involved in research on computer architecture. Zhao Wei has a Master's degree and is an engineer. He is involved in research on computer architecture. Wang Jian is a Ph.D. student.

References

[1] M. Annaratone, et al., Warp Architecture and Implementation, Proceedings of the 13rd [as published] International Symposium on Computer Architecture, ACM, 1985.

[2] R. P. Colwell, et al., A VLIW Architecture for a Trace Scheduling Computer, IEEE Transactions on Computers, Vol 37, No 8, 1988.

[3] J. A. Fisher, Trace Scheduling: A Technique for Global Microcode Compaction, IEEE Transactions on Computers, Vol C-30, No 7, 1981 pp 478-490.

[4] H. T. Kung, Network-Based Multicomputers: Redefined High Performance Computing in the 1990s, Proceedings of the Decennial Caltech Conference on VLSI, 1989.

[5] J. Labrousse and G. Slavenburg, A 50 MHz Microprocessor With a Very Long Instruction Word Architecture, Proceedings of 1990 IEEE International Solid State Circuits Conference, 1990 pp 44-45.

[6] M. S. Lam, Software Pipelining: An Effective Scheduling Technique for VLIW Machines, Proceedings of the Sigplan '88 Conference on Programming Language Design and Implementation, Atlanta, 1988.

[7] R. Mueller, B. Su, et al., A Case Study in Signal Processing Microprogramming Using the URPR Software Pipelining Techniques, Proceedings of the 19th Annual Workshop on Microprogramming (MICRO-19), 1986 pp 109-115.

[8] B. R. Rau et al., The Cydra-5 Departmental Supercomputer Design Philosophies, Decisions, and Trade-Offs, Computer, 1989 pp 12-35.

[9] B. Su, S. Ding, and J. Xia, URPR—An Extension of URCR for Software Pipelining, Proceedings of MICRO-19, 1986 pp 104-108.

[10] B. Su, S. Ding, J. Wang, and J. Xia, GURPR-A Method for Global Software Pipelining, Proceedings of MICRO-20, 1987.

[11] Su Bogong [5685 0130 3797], URPR—A Practical New Software Pipelining Method, JISUANJI XUEBAO [Chinese Journal of Computers], Vol 11, No 5, 1988.

VLIW Optimizing Compiler Adopting Two-Level Software Pipelining

92FE0867E Beijing JISUANJI XUEBAO [CHINESE JOURNAL OF COMPUTERS] in Chinese Vol 15, No 7, Jul 92 pp 491-498, 506

[Article by Su Bogong [5685 0130 3797], Wang Jian [3769 0494], Wu Yimin [0702 4135 3046], and Tang Zhizhong [3282 1807 1813] of the Qinghua University Department of Computer Science and Technology, Beijing*: "VLIW Optimizing Compiler Adopting Two-Level Software Pipelining"; MS received 24 Aug 90]

[Excerpts] **Abstract** This article begins by proposing a compiling technique that is capable of fully exploiting loop program instruction-level fine-grained parallelism: two-level software pipelining. This technology is based on the URPR [unrolling, pipelining, and rerolling] software pipelining algorithm, which organically integrates resource allocation and code optimization. It then describes a VLIW optimizing compiler that adopts two-level software pipelining and concludes with an example of an FFT inner loop compiling process and the results of preliminary experiments.

I. Introduction

VLIW (very long instruction word) computer technology has attracted people's attention because of its superior performance/price ratio. In less than 10 years' time, TRACE, Cydra5, Warp, and several other products have appeared. Their applications range from scientific computing to signal processing, image processing, and other fields. VLIW development experience indicates that, besides the design of the architecture itself, the main key problem is to design an optimizing compiler capable of fully exploiting instruction-level fine-grained parallelism[6,12]. Software pipelining is an effective technique for instruction-level loop optimization[2] and has been adopted in several VLIW compilers[5]. The URPR software pipelining algorithm we proposed in 1986 has rather good time benefits and space benefits as well rather low computing complexity, and other advantages[7]. Recently, we used the URPR algorithm as a basis for designing the URPR-1, which is suitable for signal processing and image processing. This is a multiprocessor architecture that can be implemented on a single chip (see the previous article regarding the URPR-1 architecture). The present article provides a preliminary description of the optimizing compiler that corresponds to the URPR-1 architecture. In this compiler, we proposed and implemented a new compiling technique that integrates resource allocation with code optimization: two-level software pipelining. [passage omitted]

III. Design Ideas and Overall Structure for the URPR-1 Optimizing Compiler

Because most signal processing applications programs are written in the C language, we started with the convenience of users and took into consideration program transplantability. We selected the C language as the source language for the URPR-1 optimizing compiler.

Besides conventional local and global compression optimization methods, the basic optimization measure for compilers is URPR software pipelining technology. In addition, to fully exploit the intrabody and interbody parallelism of loop bodies and thereby make full use of the PE layer and functional unit [FU] layer hardware parallelism in the machine architecture, the compiler adopts a two-level software pipelining technique that integrates resource allocation with code optimization. The compiler also uses the loop preprocessing algorithm and new loop body compression algorithm proposed in reference [11] to solve loop interrelationship problems, thereby further improving the optimization results of URPR software pipelining technology. The compiler applies the GURPR* algorithm proposed in reference [9] to solve global software pipelining problems, which enables the compiler to process the innermost loops that are the basic blocks of loop bodies and to process arbitrary complex loops in loop bodies that contain branches.

The URPR-1 optimizing compiler is composed of six modules, as shown in Figure 5 [not reproduced]. Like conventional compilers, its front end is not related to the machine and its input C language programs carry out lexical analysis and syntactic analysis and generate four-element intermediate codes.

The basic blocks that control the ranking of flow analysis partitioning of intermediate codes form the program flow chart and examine all loops.

Data flow analysis includes local analysis and global analysis. Local analysis constructs the DDG of all basic blocks and simultaneously carries out local optimization of traditional intermediate codes. Global analysis collects the data-related information among all basic blocks and derives all global variables, after which it carries out global optimization of traditional intermediate codes.

Two-level software pipelining composed of the three modules of level-one software pipelining, register allocation, and level-two software pipelining completes code generation and optimization. The level-one software pipelining module is composed of four sub-modules: data relational analysis, operation scheduling, URPR software pipelining, and functional units (such as adders and multipliers). The level-two software pipelining module is composed of five sub-modules: data relational analysis, intrabody compression, URPR software pipelining, duplicate resource (such as the read/write port of each PE) and public resource (such as busses) allocation, and loading and emptying part construction.

Register allocation in the URPR-1 optimizing compiler is different from register allocation in traditional compilers. The concern in register allocation in traditional compilers focuses on reducing the number of accesses of internal memory in object codes. Although register allocation in the URPR- 1 optimizing compiler also solves this problem, even more important problems are: 1) Allocating a group of registers in the pipeline register file to distribute those constant value operations and reference operations in different PEs. This group of registers constructs a register chain and inserts data transmission operations among the PEs; 2) The pipeline chain in the pipeline register file is used for intermediate code assignment statements. The register allocation module uses information generated by level-one software pipelining for effective resolution of these two problems.

IV. Examples

We used 1024 base-2 complex FFT innermost layer loop butterfly operations as examples of the operation of the URPR-1 optimizing compiler (see Figure 7 in the article "VLIW Architecture Based On Software Pipelining Technology" above) [not reproduced].

V. Experiment and Discussion

The prototype of the URPR-1 optimizing compiler has been implemented in a SUN-3 workstation. We used this prototype to conduct some experiments on inner loops in typical signal processing and image processing programs. The results of the initial experiments given in Table 1 show that the optimization time benefits and space benefits of the URPR-1 optimizing compiler both approximate manual coding levels. This is the result of adopting the URPR algorithm and two-level software pipelining technique. In addition, its computing complexity is $O(m^2)$, where m is the number of statements in the inner loop. Table 2 shows that the compiling time of the optimizing compiler is acceptable.

There are differences between the URPR-1 optimizing compiler and other VLIW optimizing compilers. Table 3 compares the URPR-1 compiler with the Warp compiler and Bulldog compiler.

Table 1. Inner Loop Optimization Results For Several Typical Signal/Image Processing Algorithms for the URPR-1 Optimizing Compiler (Execution time (number of cycles)/Space taken up (number of instructions))

Algorithm	Sequence code	Optimizing compiler output code	Manual coding
FFT	20n/20	$(n + K_1)/1$	$(n - K_1)/1$
Convolution and correlation	53n/53	$(n + K_2)/1$	$(n + K_2)/1$
FIR filtering	35n/35	$(n + K_3)/1$	$(n + K_3)/1$
LATTICE filtering	30n/30	$(n + K_4)/1$	$(n + K_4)/1$
IIR filtering	33n/33	$(2n + K_5)/2$	$(2n + K_5)/2$
LPC [linear predictive coding] coding	23n/23	$(2n + K_6)/2$	$(2n + K_6)/2$
3 X 3 Laplacian edge detection	79n/79	$(4n + K_7)/4$	—
Computing Gradient (using 9 X 9 canny operator)	112n/112	$(4n + K_8)/4$	—

n: Number of loops. K_1, K_2, K_3, K_4, K_5, K_6, K_7, and K_8 are constants unrelated to n and represent the execution time for the loading and emptying part

Table 2. Inner Loop Compiling Times For Several Typical Signal/Image Processing Algorithms For the URPR-1 Optimizing Compiler in a SUN-3 Workstation

Algorithm	Compiling time (s)
FFT	5.7
Convolution and correlation	7.7
FIR filtering	4.8
LATTICE filtering	4.6
IIR filtering	4.7
LPC coding	3.9
3 X 3 Laplacian edge detection	10.6
Computing Gradient (using 9 X 9 canny operator)	14.8

Table 3. Comparison of URPR-1 Compiler With Other Compilers

Compiler	Is two-level hardware parallelism partitioned?	Optimizing technique used		Are phases integrated?
		PE level	FU level	
URPR-1 compiler	Yes	Software pipelining	Software pipelining	Yes
Warp compiler	Yes	User develops coarse-grained parallelism	Software pipelining	No
Bulldog compiler	No	Loops opened and path scheduling method used for compression		Yes

*This topic was funded by the National Natural Science Fund.

References

[1] A. Aiken and A. Nicolau, Perfect Pipelining: A New Loop Parallelization Technique, Research Report, 87-873, Department of Computer Science, Cornell University, 1987.

[2] A. E. Charlesworth, An Approach to Scientific Array Processing: The Architecture Design of the AP-120B/FPS-164 Family, Computer, No 9, 1981 pp 18-27.

[3] J. R. Ellis, Bulldog: A Compiler for VLIW Architectures, The MIT Press, Cambridge, Mass., 1985.

[4] F. Gasperoni, Compilation Techniques for VLIW Architectures, Technical Report 435, New York University, 1989.

[5] M. S. Lam, Software Pipelining: An Effective Scheduling Technique for VLIW Machines, Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation, Atlanta, 1988.

[6] R. Mueller, B. Su, et al., A Case Study in Signal Processing Microprogramming Using the URPR Software Pipelining Techniques, Proceedings of the 19th Annual Workshop on Microprogramming (MICRO-19), 1986 pp 109-115.

[7] B. Su et al., URPR—An Extension of URCR for Software Pipelining, Proceedings of MICRO-19, 1986.

[8] B. Su, S. Ding, J. Wang, and J. Xia, GURPR—A Method for Global Software Pipelining, Proceedings of MICRO-20, 1987.

[9] Su Bogong [5685 0130 3797] et al., GURPR*—A New Global Software Pipelining Method, 4th National Distributed System and Firmware Engineering Academic Conference, 1990.

[10] B. Su, J. Wang, Z. Tang, W. Zhao, and Y. Wu, A Software Pipelining Based VLIW Architecture and Optimizing Compiler, Proceedings of MICRO-23, 1990.

[11] B. Su and J. Wang, Loop-Carried Dependence and the General URPR Software Pipelining Approach, Proceedings of the 24th Hawaii International Conference on System Science (HICSS-24), 1991.

[12] R. F. Touzeau, A Fortran Compiler for the FPS-164 Scientific Computer, Proceedings of the ACM SIGPLAN Symposium on Compiler Construction, 1984 pp 48- 57.